Proyecto de Sistemas Informáticos.
Curso 2005-2006.

CALIDAD DE SERVICIO EN PROCESADORES CON MULTITHREADING SIMULTANEO (SMT)

Componentes del grupo:

- Alejandro Alonso Fernández
- Noelia Morón Tabernero
- Juan Carlos Sáez Alcaide

Director del proyecto:

- Manuel Prieto Matías (DACYA)

Facultad de Informática.
Universidad Complutense de Madrid.

Autorización

Madrid, 5 Julio 2006

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

- Alejandro Alonso Fernández
- Noelia Morón Tabernero
- Juan Carlos Sáez Alcaide

Agradecimientos

Agradecemos desde aquí a los compañeros del DACYA que amáblemente han colaborado con nosotros y también a los que nos han aguantado día tras día en el laboratorio; en especial a Enrique de la Torre, Edgardo Mejía y Jose Ignacio Gómez.

De manera muy especial, por su desinteresada ayuda y paciencia a Javier Setoain, que ha respondido sin quejarse a todas nuestras dudas.

Y por supuesto, a nuestro director de proyecto Manuel Prieto Matías por guiarnos y motivarnos.

Lista de palabras clave

Planificador
Calidad de Servicio *(qos)*
Kernel
Linux
Contadores Hardware
Rendimiento
Multithreading Simultaneo *(SMT)*

Resumen

SMT: Implementación de un algoritmo para dar Calidad de Servicio

Aunque en general, con hyperthreading (mecanismo cuya base es tener varios procesadores lógicos sin tener todo el hardware duplicado) se consigue mayor productividad, dicha mejora puede conseguirse a expensas de disminuir los recursos disponibles para procesos críticos.

Experimentos previos indican que para conseguir políticas que maximicen el número de plazos cumplidos por tareas con requisitos de tiempo real suave, no basta con la asignación de prioridades tradicional de Linux, sino que es necesario tener en cuenta SMT.

Resulta interesante, por tanto, estudiar el efecto de una **planificación basada en calidad de servicio** que optimice el rendimiento de una tarea sin degradar la respuesta del resto de procesos ejecutándose en el sistema.

Nuestra propuesta de calidad de servicio se ha llevado a cabo para procesadores de la familia *Pentium 4 e Intel Xeon* y sobre el kernel de Linux para la familia 2.6; cuyos mecanismos para calidad de servicio para este tipo de procesadores consideramos no están suficiéntemente estudiados.

SMT: Implementing an algorithm for Quality of Service

Although normally, with hyperthreading (mecanism based in having various logical processors without having all the software duplicated), bigger throughput is obtained; this improvement can force a reduction in the available resources for critical processes.

Previous experiments has shown that obtaining policies which maximize the number of carried out term's by task with real time requirements, is not enough the traditional linux priority asignation, it's necesary taking account of SMT.

So it's interesting estudying the effect of Quality of Service based policy that optimizes one task's throughput without affecting other processes' response time in the system.

Our Quality of Service proposal has been developed for *Pentium 4 e Intel Xeon* family processors and over the kernel 2.6 series. To the best of our knowledge, scheduling mechanisms to obtain Quality of Service for this kind of processors hasn't been yet implemented on real systems.

Índice general

Índice de figuras

Índice de cuadros

Parte I

Introducción

Capítulo 1

Introducción

Uno de los objetivos del planificador de procesos en un sistema operativo multiprogramado, es lograr obtener la mayor productividad y el máximo aprovechamiento del procesador. Por otro lado, el soporte para ofrecer calidad de servicio (QoS) constituye otra caracterísica deseable del sistema.

La calidad de servicio, en el contexto de la planificación de procesos, consiste en basar la toma de decisiones de planificación en función de las distintas prioridades de los procesos, y así lograr favorecer a los procesos de más alta prioridad en el reparto de la CPU.

Conseguir ambos objetivos, productividad y calidad de servicio, –a priori contrapuestos– es una tarea compleja, y, en general, se prefiere una implementación del planificador que de prioridad a calidad de servicio sin que ello disminuya considerablemente la productividad del sistema.

En la implementación del planificador de Linux para arquitecturas monoprocesador y arquitecturas SMP convencionales, un proceso de una cola de ejecución que está listo para ejecutar y es más prioritario que otro[1], entrará antes a ejecutar; y además se le asignará un tiempo mayor de CPU (*timeslice o quanto*) para consumir. De este modo, se asocia una mayor prioridad con un mayor *timeslice* y más preferencia en cuanto al uso inminente del procesador asociado.

Si utilizamos esta estrategia para ofrecer QoS en una arquitectura con SMT, estamos tratando los procesadores lógicos en los que está dividido un procesador físico,

[1] En sistemas SMP, el planificador agrupa los procesos en colas de ejecución. Existe una cola de ejecución por procesador. El planificador realiza la planificación sobre las colas de procesos, de manera independiente. Siguiendo esta estrategia, se identifica al proceso más prioritario como el más prioritario de su cola. En arquitecturas monoprocesador, solo hay una cola de ejecución ya que sólo hay un procesador.

como procesadores o "cores" independientes. Sin embargo, estos procesadores lógicos, comparten gran cantidad de recursos hardware como unidades funcionales, niveles de memoria cache ("*on chip*"), buffer de reordenamiento, hardware de predicción de saltos,... Por tanto, la planificación de colas de ejecución independientes no garantiza que el proceso más prioritario del sistema, esté recibiendo el trato más prioritario; ya que puede estar compitiendo por recursos compartidos del procesador, con otro proceso de menor prioridad, que esté ejecutando en otro procesador lógico.

La estrategia que hemos seguido en nuestro proyecto, persigue preservar la calidad de servicio del planificador de Linux sin afectar sobremanera a la mejora de productividad que se consigue con la arquitectura SMT. Para ello alternamos situaciones con el SMT activado, con otras en las cuales esta característica está desactivada.

Con la característica SMT activada conseguimos mayor productividad, ya que disponemos de n unidades de ejecución; mecanismo que ofrece gran capacidad de ocultación de latencias originadas por accesos al sistema de memoria. Por otra parte, para preservar la calidad de servicio, debemos ofrecer al proceso más prioritario, suficientes recursos hardware para garantizar su trato preferente. Cuando el *Simultaneous Multithreading* está desactivado, todos los recursos pasan a estar completamente disponibles para una unidad de ejecución; dando de este modo soporte para el trato preferente a un proceso determinado.

Para detectar, la *disputa* por los recursos entre procesos, que provoca la alteración del tratamiento preferente dado al proceso más prioritario, es preciso monitorizar su comportamiento en ejecución. Gracias a los contadores de monitorización del rendimiento (*Performance Monitoring Counters, PMC's*) –también llamados contadores hardware y ampliamente descritos en la sección 2.1–, podemos registrar estas situaciones no deseadas de competición por los recursos. Una vez detectadas estas situaciones, actuaremos en consecuencia y pondremos todos los recursos hardware del procesador disponibles para el proceso más prioritario –deshabilitando el SMT, por una u otra vía– .

Capítulo 2

Introducción a la tecnología usada

Diversas tenologías han servido como base para el desarrollo del prototipo presentado. Es clave conocer tanto sus limitaciones como sus ventajas y por ello haremos un breve repaso de cada una de ellas, centrándonos en los aspectos relevantes que las hacen puntos clave en el estudio previo a la realización del prototipo. Principalmente hablamos de los **contadores hardware**, la tecnología **HyperThreading** y el **planificador** (estructura clave del kernel sobre la que trabajaremos).

Puesto que el trabajo sobre el kernel en sí se caracteriza en parte por la árdua labor de depuración, enumeraremos qué mecanismos se pueden usar actualmente para la misma. Finalmente, repasaremos las aplicaciones existentes que tienen alguna relación con el trabajo realizado y los motivos por los cuales hemos descartado su funcionalidad.

2.1. Contadores hardware

Los **contadores hardware** son un conjunto de **registros** del procesador que tienen la propiedad de poder contar los sucesos que se desee. Basta con hacer unas configuraciones previas sobre ellos y estos registros contarán el evento para el que han sido configurados, como por ejemplo instrucciones que se ejecutan en el procesador ó fallos de caché.
Cada registro contador tiene asociados otros dos registros que sirven para configurar detalles relativos a la cuenta del evento deseado. Estos registros se llaman ESCR y CCCR.
A grandes rasgos el funcionamiento de un registro contador consiste en cargar con los valores adecuados sus registro ESCR y CCCR, ordenar la iniciación de la cuenta y leer su contenido cuando se desee, existiendo la posibilidad de resetearlo o parar la cuenta.

La naturaleza de estos contadores varía según la arquitectura sobre la que se esté trabajando. Puesto que el uso de estos contadores no es en absoluto sencillo se ha realizado una interfaz que nos abstrae por completo de su complejidad. En concreto, esta interfaz desarrollada se ha hecho para la arquitectura $IA-32$ de la familia *Pentium 4 e Intel Xeon*.

2.2. Contadores hardware en la familia *Pentium 4 e Intel Xeon*

El mecanismo de monitorización de eventos suministrado en la familia *Pentium 4 e Intel Xeon* nos proporciona las siguientes opciones:

- Un conjunto de eventos predefinidos y métricas asociadas que simplifican la configuración y puesta en marcha de los contadores hardaware.

- Selección de eventos mediante unos registros denominados *ESCR's*. Estos registros son limitados (de 43 a 45 dependiendo de la familia y el modelo) y se han de cargar con valores adecuados (máscaras) que especifiquen correctamente qué eventos han de ser controlados para incluirse o no en el conteo.

- Registros MSR contadores en los que se almacena la cuenta de los eventos. Cada microprocesador dispone de 18 contadores de este tipo.

- Selección del control de los eventos seleccionados. Mediante un registro adicional denominado CCCR asociado al MSR correspondiente se establece el estilo o tipo de conteo que se lleva a cabo.

- El *IA32_MISC_ENABLE MSR*, que indica la disponibilidad en los procesadores $IA-32$ de la opción de monitorizar el rendimiento y la existencia de la posibilidad de medir eventos en modo preciso (*PEBS*).

- El *IA32_PEBS_ENABLE MSR* que habilita el conteo y etiquetado de los eventos de un tipo especial denominados *at-retirement*.

2.2.1. Eventos que se pueden monitorizar

Calsificación según el momento en que sucede el evento

Podríamos hacer una clasificación de los eventos que pueden ser monitorizados en dos grupos:

- Eventos *Non-retirement*. Se trata de eventos que ocurren durante la ejecución de una instrucción tales como operaciones con intervención de la caché.

- Eventos *At-retirement*. Son eventos que ocurren en la fase de retirada de la instrucción, puesto que en ella se puede capturar el estado final tras la ejecución de la instrucción del máquina.
  Este tipo de conteo incluye la posibilidad de marcar μop que pueden haber sido seleccionadas como evento a ser medido a lo largo de la ejecución de la instrucción. Esto proporciona dos modos de uso. El marcaje o etiquetado permite distinguir entre los eventos que ocurren durante la traza de ejecución y no llegan a la fase de *commit* debido por ejemplo a saltos mal predichos y los eventos que sí se producen en fase de *commit*.

Calsificación según el modo de monitorizar

La familia *Pentium 4 e Intel Xeon* proporciona tres modos de uso de la monitorización. Los dos primeros que se describen a continuación se pueden usar tanto para contar eventos *non-retirement* como eventos *at-retirement*. El tercer modo solo sirve para eventos *at-retirement*.

- *Event counting.* Un contador se configura para medir uno o más tipos de eventos. Mientras el contador está contando, el software lee su contenido en intervalos de tiempo pre-establecidos por el número de veces que ha ocurrido el evento entre los intervalos.

- *Non-precise event based sampling.* Un contador se configura para medir uno o más tipos de eventos y para generar una interrupción cuando este se desborda. Estas interrupciones se denominan *PMI (performance monitoring interrupt)*. La rutina de tratamiento de esta interrupción salva el *RIP (return instruction pointer)* y resetea el contador entre otras acciones.

- *Precise event-based sampling (PEBS).* Este tipo es similar al anterior pero ahora se usa un buffer de memoria para salvar el estado de la arquitectura cuando el contador se desborda.

Calsificación según el procesador lógico en que tienen lugar

Es necesario dejar clara una ultima distinción acerca la naturaleza de los ventos que se pueden medir. Puesto que estamos hablando de monitorización en procesadores con hyper-threading, resulta interesante el poder distinguir si un evento se lleva a cabo en un procesador lógico o en otro. Una clasificación basada en la posibilidad de establecer esa distinción nos lleva a distinguir entre dos grupos de eventos:

- *Thread specific TS* El evento puede configurarse para ser medido en un procesador lógico específico.

- *Thread independent TI* El evento no puede caracterizarse para ser medido en un procesador lógico distinguido. su medición se realizará teniéndo en cuenta ambos procesadores, ignorando cualquier configuración del ESCR que diga lo contrario.

2.2.2. Selección del evento a monitorizar: Carga del registro *ESCR*

Un registro ESCR se configura para detectar un evento que se cuenta en un registro contador. La caracterización del evento a monitorizarse depende del valor de los campos que a continuación se describen:

- Los 7 bits del campo *event select* seleccionan el evento a medir y los 16 bits del campo *event mask* seleccionan un subconjunto distinguido de los eventos deseados.
  Por ejemplo, para el evento *branch_retired* determinariamos una única máscara para el campo *event select*. Pero a su vez podriamos distinguir 4 tipos de saltos: *taken, not taken , predicted* y *mispredicted* y según el que se escoja varía la máscara del campo *event mask*.

- Mediante el campo *tag value* se establece el valor de la etiqueta que se usará para los eventos *at-retirement*. Para ello necesita estar activado el flag correspondiente al campo *tag enable* (habilitación del tagging de μop).

- Los cuatro bits menos significativos de un registro ESCR permiten escoger si contar el evento cuando ocurre en modo usuario, o en modo sistema y cuando ocurre en un procesador lógico o en otro. Es decir, permite distinguir entre los dos hilos de vía de ejecución concurrente existentes en el Pentium 4 con SMT y el modo de privilegio en el que ha de ser medido.
  Por ejemplo, para medir todos los eventos del sistema operativo que ocurren en los dos hilos, se activarian los flags *T0_OS* y *T1_OS* y para medir los eventos de usuario para el procesador lógico 0 y los de sistema para el 1 deberian activarse *T0_USR* y *T1_OS*.
  Hay que tener en cuenta lo comentado anteriormente sobre la naturaleza TI o TS de los eventos. Por ejemplo, para un evento TS (*thread specific*) que se produce en el procesador lógico 0, su cuenta depende solo de haber configurado los bits *T0_USR* y *T0_OS* de su ESCR. Pero para un evento TI la medición se hace en los dos procesadores y solo podriamos escoger el modo de lacuenta: si usuario o sistema. Por ejemplo, para una configuración *T0_USR=1*, *T0_OS=0*, *T1_USR=0*, *T1_OS=0* se contaria el evento en ambos procesadores cuando ocurriese en modo sistema.

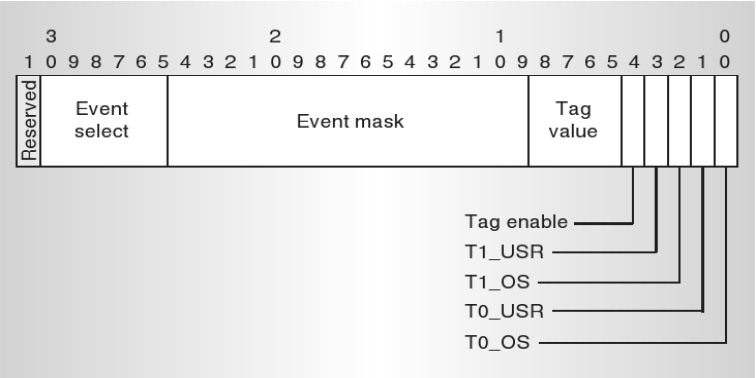


Figura 2.1: Registro de selección de evento (ESCR) en la familia *Pentium 4 e Intel Xeon*

2.2.3. Selección del modo de monitorización: Carga del registro *CCCR*

Una vez seleccionado el evento que se desea monitorizar, es necesario establecer en qué condiciones ha de hacerse el conteo del mismo. Esto ha de indicarse mediante los valores de los campos que se cargan en el registro CCCR asociado al evento.
Las acciones de control que se indican con la máscara propuesta para un determinado CCCR son las siguientes:

- Habilitar o desabilitar el contador.
- Seleccionar el detector del evento que se usará como fuente para los incrementos en el contador. Es decir, identifica el ESCR que se usará para seleccionar el evento
- Posibilidad de filtrar el contador con umbrales, complementos etc..Se trata de un bit denominado flag de comparación cuya activación o desactivación posibilita el filtrado.
- Establecer el umbral de la cuenta y su modo de uso (flag de complemento). Si el bit de comparación está activado, y el flag de complemento también, el campo *threshold field* establece el umbral según el cual los valores menores o iguales a él no se tienen en cuenta en el conteo. Si el bit de comparación está activado, y el flag de complemento no, los valores que se devuelven son los mayores.
- Activar o desactivar la salida de comparación habilitando el flag *Edge flag* siempre y cuando el flag de comparación esté activo.

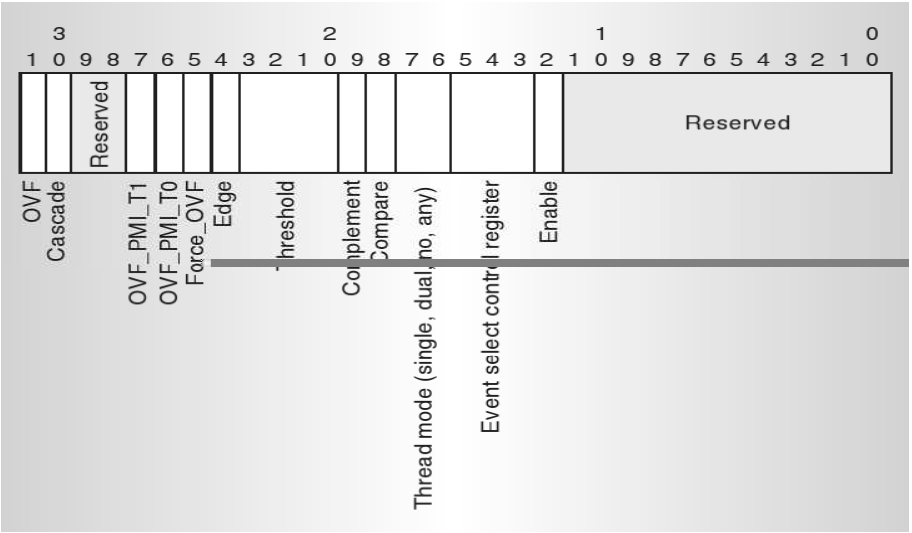


Figura 2.2: Registro de configuración (CCCR) en la familia *Pentium 4 e Intel Xeon*

- Opciones sobre situaciones dedesbordamiento:

  - Con el flag *FORCE_OVF* la señal de desbordamiento se activa en cada incremento del contador

  - Mediante el flag *OVF_PMI* se lanza una interrupción PMI cada vez que haya un desbordamiento.

  - Activando el flag *cascade flag* hace que los contadores funcionen en cascada alternandose con otros en situaciones de desbordamiento.

  - Si el flag *OVF* está activo indica que ha habido un desbordamiento.

2.2.4. Registros en los que se hace la cuenta: Performance counters

Los contadores, junto con los registros de configuración CCCR se usan para filtrar y contar los eventos seleccionados por los registro ESCR.
Los procesadores de la familia *Pentium 4 e Intel Xeon* disponen de 18 contadores organizados en nueve pares. Cada par de contadores está asociado a un subconjunto concreto de eventos y con ello a un subconjunto de ESCR's.
Para saber qué eventos se pueden contar con cad par hay que consultaren la documentación la Tabla 15-6 de [8].

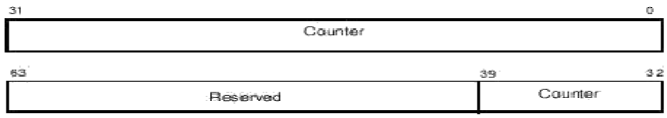


Figura 2.3: Contador de rendimiento en procesadores de la familia *Pentium 4 e Intel Xeon*

Cada contador tiene 40 bits, como se puede ver en 2.5, lo cual proporciona un límite de cuenta bastante amplio que reduce las posibilidades de desbordamiento.

La instrucción *RDPMC* permite leer estos registros. Una de las peculiaridades de los Pentium 4 y Xeon, como ya se ha dicho antes es que permiten la lectura rápida de contadores. Esta lectura rápida consiste en leer los 32 bits menos significativos de estos registros. Esto es útil cuando se sabe de antemano que el evento que se va a contar no va a exceder el valor máximo y por tanto no va a producir un desbordamiento del contador.
La manipulación de los contadores sólo la puede llevar a cabo el sistema operativo ejecutándose en nivel de privilegios 0, mediante las instrucciones *RDMSR* y *WRMSR*. Un sistema operativo seguro debe limpiar el flag PCE (relacionado directamente con el nivel de privilegios) durante la inicialización del sistema y desactivar el acceso a los usuarios a los contadores hardware, y a la vez proveer a los usuarios de una interfaz que permita emular la instrucción *RDPMC*.
Para escribir en los contadores se utiliza la instrucción *WRMSR* con la que se configuran los contadores.

2.2.5. Programación de los contadores hardware para medir un evento

Medición de eventos simples (*Non-Retirement*)

Para poder medir los eventos hay que seguir una serie de pasos:

- Para un evento que se desee medir, seleccionar un ESCR compatible con él comprobando las restricciones. Esto se puede ver en la Tabla A-1 de [8].

- Seleccionar el contador CCCR complatible a partir de los valores de ESCR.
- Establecer los niveles de privilegios que se van a monitorizar en el ESCR.
- Activar el conteo en cascada si se desea.
- Opcionalmente configurar el CCCR para generar interrupciones PMI cuando el contador desborda. El APIC local debe estar preparado.
- Comenzar el conteo del evento.

Conseguir los valores con este método es algo tedioso, con lo que se buscó otra forma para conseguir estos valores y configuraciones. Para ello se utilizó la aplicación presentada *Brank*.

Medición de eventos que necesitan tagging (*At-Retirement*)

Medir eventos del tipo at-retirement significa medir solamente eventos que pueden darse en la fase de commit, ignorando por tanto todo el trabajo de especulación que lleva a cabo el procesador. Mediante at-retirement se pueden monitorizar eventos que préviamente han sido etiquetados (tagging) y que son seleccionados por dicha etiqueta en la fase commit.
Las acciones especulativas (por ejemplo la predicción de saltos) se llevan a cabo para incrementar la efectividad y elevar el rendimiento. Los Pentium 4 y Xeon llevan el típico predictor de saltos según la dirección de estos y después decodifican y ejecutan la instrucción predicha, anticipándose al cálculo de la dirección real. Cuando se produce un fallo en la predicción, los resultados de las instrucciones que han sido decodificadas y ejecutadas por la rama del fallo de predicción, se cancelan.

Relativo a este tema, Intel proporciona una serie de definiciones que permite clasificar las intrucciones:

- **Instrucciones Bogus, Non-Bogus y retired**. Con *Bogus*, Intel se refiere a las instrucciones o *µops* que deben ser canceladas porque están en una rama de ejecución en la que se ha producido un fallo de predicción de salto.
  Con *Non-Bogus* y *retired* se refiere a las instrucciones o *µops* que producen cambios

en el estado de la arquitectura. Las instrucciones pueden ser retired o Non-Bogus pero nunca las dos a la vez.

- **Etiquetado Tagging**. Consiste en marcar *μops* relacionadas con un evento particular que se quiere monitorizar.
  Se contará el numero de instrucciones cuando sean retiradas. Durante la ejecución, se puede producir el evento mas de una vez por *μops*, con lo que no es del todo exacto, ya que al final se miden las instrucciones y no el evento directamente.
  El etiquetado permite que una *μop* sea etiquetada una vez y esta mantendra la etiqueta el tiempo en el que está en el pipe, para ser contada al ser retirada. Se utiliza para métricas de rendimiento que se incrementan en uno por cada *μop* retirada.

- **Replay**. Para maximizar el rendimiento en las actividades más comunes, la microarquitectura NetBurst hace una planificación *agresiva* para poder ejecutar *μops* antes de que se tengan garantías de la correcta ejecución. Cuando no se garantiza alguna de las condiciones necesarias para la ejecución de la *μops*, se produce un evento denominado *relanzamiento* (reissued).
  Para poder relanzar la *μop* los procesadores Pentium 4 y Xeon utilizan el mecanismo replay. Algunos ejemplos de replay son los fallos de predicción y violaciones de dependencias. En operaciones normales, algunos de los replays producidos son comunes e inevitables. Un excesivo numero de replays indica que se está produciendo un problema de rendimiento.

- **Assist**. Cuando el hardware necesita ayuda con el microcódigo, la máquina le proporciona lo que se denomina *asistencia*. Por ejemplo, cuando no han llegado los operandos necesarios para una operación de punto flotante. El hardware debe internamente modificar el formato de los operandos para que no se degrade el rendimiento.

Medición de eventos pebs (*Precise Event-Based Sampling*)

El área de depuracion se introdujo en la familia *Pentium 4 e Intel Xeon* para permitir recoger diferentes tipos de información en los buffer residentes en memoria y poder emplearla en la depuración y modificación de aplicaciones.
Dicha información se salva en el área DS.

El DS de los Pentium 4 y Xeon permite recoger dos tipos de información para usarlos en depuración y optimización de programas. Estos son los registros PEBS y los BTS.

Los BTS o Branch Trace Store son las trazas de saltos tomados, interrupciones y excepciones.
PEBS permite salvar el estado preciso de la arquitectura asociada con uno o más eventos. Esta información se guarda en un buffer que es una parte asignada del DS. Para usar este mecanismo, un contador tiene que ser configurado para permitir lanzar el desbordamiento. Cuando un contador desborda, el procesador copia el estado actual de los registros de propósito general, los registros EFLAGS y el contador de programa en un registro en el buffer situado en el DS. Después el procesador borra el contador y comienza a contar de nuevo. Cuando el buffer está casi lleno, se genera una interrupción, permitiendo que los registros sean salvados. No se trata de un buffer circular, para evitar que se sobreescriban datos.
Solamente soportan PEBS un determinado subconjunto de eventos *at-retirement* como son: Execution event, Front end event, y Replay event.
Para activar PEBS, el procesador debe soportarlo. Esto se puede observar viendo el resultado de ejecutar la instrucción *CPUID*. Si está permitido, entonces son accesibles los siguientes campos:

- El flag PEBS UNAVAILABLE en el registro IA32 MISC ENABLE MSR que indica si está habilitado PEBS (cuando está a 0).
- El flag de activación de PEBS (bit 24) en el IA32 PEBS ENABLE MSR permite activar o desactivar PEBS.
- El IA32 DS AREA MSR, que puede ser programado para que apunte al DS.

2.3. HyperThreading: *SMT* en las familias Pentium 4 e Intel Xeon

Actualmente existen diversas alternativas en el contexto de la explotación del paralelismo en un computador. Principalmente han cobrado importancia dos: los diseños basados en varios procesadores en un chip (*MoC*)y el diseño fundamentado en un único procesador que permite la ejecución simultánea de varios hilos (*Multithreading Simultáneo* ó *SMT*).

La primera implementación de SMT en el mercado de procesadores de propósito general se denominó *Hyper-Threading* (NT) y fué introducida en el procesador *Intel Xeon MP de Intel*.
En la actualidad, una evolución de dicha implementación, se utiliza en las familias *Intel Pentium 4* e *Intel Xeon*.

En las descripciones que plantea *Intel*, presenta esta tecnología como una técnica que interpreta el **procesador físico** como **dos procesadores lógicos**.
Para ello se *duplica* el *estado arquitectónico del procesador* (registros de propósito general, registros de control y registros del control de interrupciones), mientras que el *resto de recursos* se utilizan de forma *compartida*. De hecho, en el caso de *Intel Xeon* solo fueron necesarios un 5 % más de transistores.

2.3.1. Alternativas Multithreading existentes

Como se ha explicado anteriormente, las arquitecturas multithreading se caracterizan por mantener activos varios flujos de ejecución a la vez en un mismo procesador, pudiendo ser estos de una misma aplicación o de aplicaciones diferentes. Siguiendo este criterio se obtienen varias vertientes definidas por cada una de las siguientes arquitecturas:

- ***FMT* ó Multithreading de Grano Fino**: dentro del pipeline hay activas instrucciones de hilos de ejecución distintos. Así, aunque no se llegan a lanzar simultáneamente instrucciones en el mismo ciclo, se oculta con trabajo últil de un hilo las latencias del otro, debido a operaciones como por ejemplo los accesos a memoria.

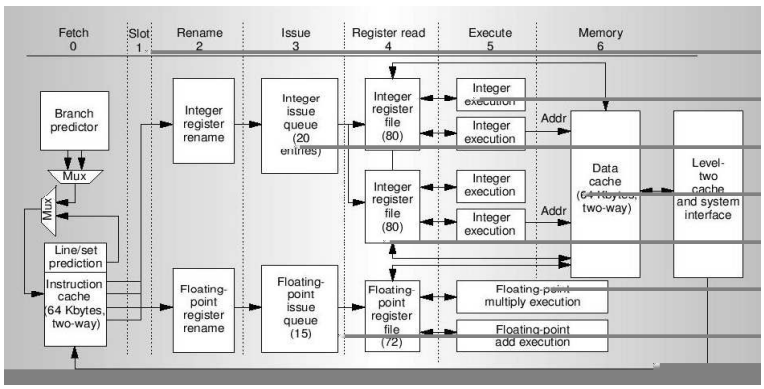


Figura 2.4: Pipeline Alpha 21464

- ***BMT* ó Multithreading de Grano Grueso**: los cambios de contexto no se producen cada ciclo sino ante la ocurrencia de eventos muy costosos que puedan provocar largas paradas en el pipeline.

- ***SMT* ó Multithreading simultáneo**: su característica fundamental es que permite el lanzamiento de instrucciones (issue) provenientes de varios contextos hardware. El multithreading simultaneo el la alternativa multithread que más se ha comercializado.

2.3.2. Multithreading Simultaneo (SMT)

Tanto el multithreading de grado fino como el de grano grueso se pretenden limitar las latencias mediante cambios de contexto.
SMT sin embargo puede considerarse como una evolución de las arquitecturas superescalares cuya intención fundamental es la conversión de paralelismo a nivel de thread en paralelismo a nivel de instrucción. Ello genera una mayor utilización de los recursos del procesador y consecuentemente, se reducen pérdidas de rendimiento por latencias excesivas.

La introducción estable del SMT en la industria llegó en el año 2002 con la incorporación de los procesadores *Intel Pentium 4* e *Intel Xeon*. También se incorporó en la familia *Power*, concretamente en el modelo *Power 5* de IBM.

Organización de los recursos del procesador SMT

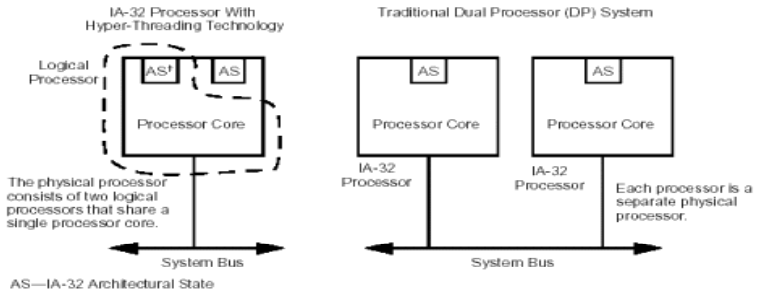


Figura 2.5: Comparación de un procesador con Hyper-Threading y un sistema con dos procesadores

Como se comentado anteriormente, los recursos de un procesador pueden organizarse de forma compartida entre los prcesadores lógicos o bien duplicarse.

Organización compartida

En la organización compartida, los recursos compartidos son :

- Buffer de instrucciones ó IFB (Instruction Fetch Buffer).
- Banco/bancos de registros.
- Ventana de instrucciones.
- Buffer de reordenamiento ó ROB (Reorder Buffer).

Para poder distingur de qué procesador lógico es cada instrucción basta con añadir etiquetas a las intrucciones y añadir más registros físicos para reducir los riesgos estructurales. También es necesario una modificación en las etapas de *fetch* y *retire* (respectivamente búsqueda y retirada de instrucciones) para permitir que puedan hacerse de modo concurrente.

Organización replicada

Los buffer internos del procesdor han de duplicarse. También las unidades de búsqueda, decodificación, renombrado de registros y retirada de instrucciones deben duplicarse o bien multiplexarse como en la orgranización compartida.
Las modificaciones necesarias son más complejas que en la versión compartida. Sin embargo, compensan altamente porque la replicación de la ventana de instrucciones y del

ROB simplifican notablemente el lanzamiento y la retirada de instrucciones.

2.4. Planificador en la versión *2.6.13*

Linux es un sistema muy complejo, compuesto de muchos componentes divididos en capas, relacionados entre sí. Estos componentes se comunican, se sincronizan y funcionan de manera concurrente, lo que hace que un sistema ya de por sí complejo por su propia extensión, se convierta en algo en lo que las consecuencias de pequeños cambios en sistemas clave se hagan difíciles de predecir. Dentro del núcleo existe una parte central que articula los mecanismos principales del sistema, que llamamos los subsistemas del kernel (gestor de memoria, sistema de ficherosvirtual, subsistema de comunicación de procesos, subsistema de red y el planificador).

La misión del planificador de un sistema operativo es decidir cómo, cuándo y dónde se ejecutarán lastareas que esán activas en el sistema.

El planificador, en la rama de Linux *2.6.x*, ha sido reescrito entero e incorpora grandes cambios respecto a versiones anteriores, las cuales han mejorado mucho sus capacidades, rendimiento y escalabilidad. Algunos de los cambios más significativos en el planificador, como la complejidad en O(1) o la expropiación, son explicados en detalle a lo largo de este capítulo.

2.4.1. Estructura del Planificador

El planificador de Linux está implementado en *kernel/sched.c*. El algoritmo fue reescrito por completo en la versión *2.5* y, al contrario que en las versiones anteriores, fue diseñado para cumplir objetivos específicos:

1. Complejidad O(1): cada algoritmo en el nuevo planificador debe finalizar en tiempo *constante*, independientemente del numero de procesos que haya corriendo en el sistema.

2. Escalabilidad *perfecta* en SMP: cada procesador debe tener su propia cola de procesos (runqueue) y su lock de protección asociado. De esta manera es posible que de forma simultánea, dos tareas se despierten en distintas CPUs, sean planificadas y se hagan los correspondientes cambios de contexto.

3. Afinidad SMP mejorada: el planificador debe intentar agrupar y mantener a los procesos en una CPU específica. Sólo se migrarán a otra CPU para conseguir un

mejor equilibrado de carga. Así, se impide que haya tareas que estén continuamente siendo migradas de una Cola de procesos a otra.

4. Eficiencia SMP: ninguna CPU debe estar inactiva si hay tareas que ejecutar.
5. Planificación por lotes: timeslices largos y polótica Round-Robin. Se aplica a las tareas con menor prioridad, es decir, con un valor de nice positivo.
6. Ejecutar los procesos hijos recién creados antes que sus procesos padres.
7. Buen rendimiento con tareas interactivas: incluso con una carga considerable, el sistema debería reaccionar y planificar las tareas interactivas inmediatamente. Esto beneficia enormemente a sistemas tipo desktop.
8. Optimizado para el caso de una o dos tareas en ejecución.
9. Reparto justo de los timeslices: ningun proceso debe sufrir inanición ni tener de manera injustificada un timeslice demasiado alto.

A continuacion veremos las principales estructuras de datos presentes en el planificador:

Procesos

Toda la información relativa a los procesos se guarda en su Descriptor de Proceso. Esta información incluye ficheros abiertos, los datos de su espacio de memoria, señales pendientes, estado del proceso, etc. En Linux, la información del descriptor de procesos se guarda en la estructura task_t 2.4.1 que está definida en *include/linux/sched.h*. A diferencia de otros sistemas operativos, en Linux son totalmente equivalentes los conceptos de procesos e hilos de ejecución (threads). Otra peculiaridad de Linux, es que emplea el nombre de tareas (tasks) para referirse a los procesos. Por tanto se utilizarán ambos términos de manera indistinta.

Estructura de los procesos

La estructura task_t posee muchos campos, pero los más importantes, en lo que a la planificación se refiere, son los siguientes:

```
typedef struct task_struct task_t;
struct task_struct {
    prio_array_t *array;
    struct list_head run_list;
    int prio;
    int static_prio;
```

```
    unsigned int time_slice;
    unsigned long sleep_avg;
    volatile long state;
    int activated;
    unsigned long rt_priority;
    unsigned long policy;
    cpumask_t cpus_allowed;
    spinlock_t switch_lock;
    struct thread_info *thread_info;
    struct sched_info sched_info;
};
```

A continuación se describen algunos de estos campos:

1. **prio_array_t *array**: arrays que albergan a todos los procesos clasificándolos por su prioridad.

2. **struct list_head run_list**: lista de procesos asociada al nivel de prioridad de la tarea. Los Arrays de Prioridad y sus listas de procesos se explicarán en la sección 2.4.3.

3. **unsigned long rt_priority**: es la representacion que ve el usuario de los niveles de prioridad de los procesos de tiempo real.

4. **unsigned long policy**: es el esquema de planificación empleado para este proceso. Tanto este, como el campo rt_priority se explican en la sección 2.4.3.

5. **int static_prio**: Prioridad Estática. Es la prioridad elegida por el usuario para los procesos que no son de tiempo real (los procesos de usuario).

6. **int prio**: es la prioridad efectiva del proceso, tanto si es de tiempo real, como si es de usuario.

7. **unsigned long sleep_avg**: tiempo medio que el proceso pasa suspendido (sleep average). Es una heurística empleada por el planificador para medir la *interactividad* de un proceso, y así, a justar su prioridad. Las prioridades de los procesos de usuario se explican en la sección 2.4.3.

8. **volatile long state**: estado del proceso. Se explica más adelante.

9. **int activated**: este campo es empleado cuando se despierta a un proceso. En función del estado en el que estuvo suspendido, se le da al campo un valor que representa su grado de interactividad. Posteriórmente, se modificará su sleep average segun este valor.

10. **int time_slice**: tiempo de ejecución del proceso. Este valor es elegido en función de su prioridad estática.

11. **cpumask_t cpus_allowed**: máscara que indica los procesadores en los que se puede ejecutar el proceso. Se emplea en los equilibrados de carga.

12. **spinlock_t switch_lock**: lock empleado en algunas arquitecturas durante los cambios de contexto.

13. **struct thread_info *thread_info**: estructura empleada para acceder de manera eficiente al descriptor de procesos. Se explica más adelante.

14. **struct sched_info sched_info**: estructura utilizada en la recolección de estadísticas a nivel de proceso.

Estados de los procesos

El estado de los procesos es descrito por el campo state de la estructura task_t (ver 2.4.1). Algunos de los estados de los procesos son los siguientes:

- TASK_RUNNING: el proceso es ejecutable y por tanto se encuentra en el Array de Prioridad, ya sea en ejecución, o esperando su turno. El valor por defecto de esta constante es 0.

- TASK_INTERRUPTIBLE: el proceso está suspendido esperando a que sea cumpla alguna condición. Cuando esto ocurra, el proceso será despertado y puesto en estado TASK_RUNNING. Sin embargo, también puede ser despertado de forma prematura si recibe alguna señal (por ejemplo,SIGTERM). Los procesos interactivos suelen estar suspendidos en este estado.

- TASK_UNINTERRUPTIBLE: igual que el caso anterior, salvo que el proceso no será despertado si recibe alguna señal.

- TASK_TRACED: el proceso está siendo depurado.

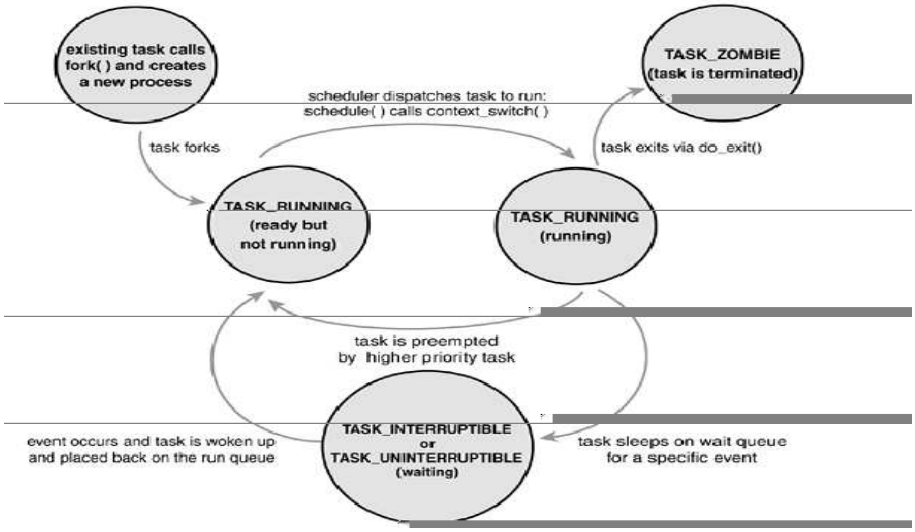


Figura 2.6: Estados de los procesos y posibles transiciones.

- TASK_STOPPED: la ejecución del proceso ha sido detenida. En este caso el proceso ya no se ejecuta ni puede ser elegido por el planificador para ejecución. Esto ocurre cuando la tarea recibe las senales SIGSTOP,SIGSTP, SIGTTIN o SIGTTOU, o bien, si se recibe cualquier señal mientras está siendo depurada.

- EXIT_ZOMBIE: el proceso ha acabado, pero su descriptor sigue en memoria por si el padre necesita alguna información. Dicho descriptor será eliminado cuando el padre ejecute la llamada al sistema wait4().

- EXIT_DEAD: el proceso ha acabado y se ha eliminado su descriptor.

El mecanismo más empleado para cambiar el estado de un proceso es utilizando las macros *set_task_state (task,state)* y *set_current_state(state)*.

La diferencia es que la segunda sí lo cambia el estado del proceso current (el que está actualmente en ejecución). En sistemas SMP, además, puede establecer una barrera para sincronizar a las tareas del resto de procesadores. Las transiciones entre estados de los procesos, y sus causas, se pueden observar en la figura 2.6 [8].

Accediendo al Descriptor de Proceso

En versiones de Linux anteriores a la 2,6, el descriptor de procesos (task_t) se almacenaba al final de la pila de sistema (kernel stack) de cada proceso, de tal manera que en arquitecturas con pocos registros, como la x86, para acceder a este descriptor se

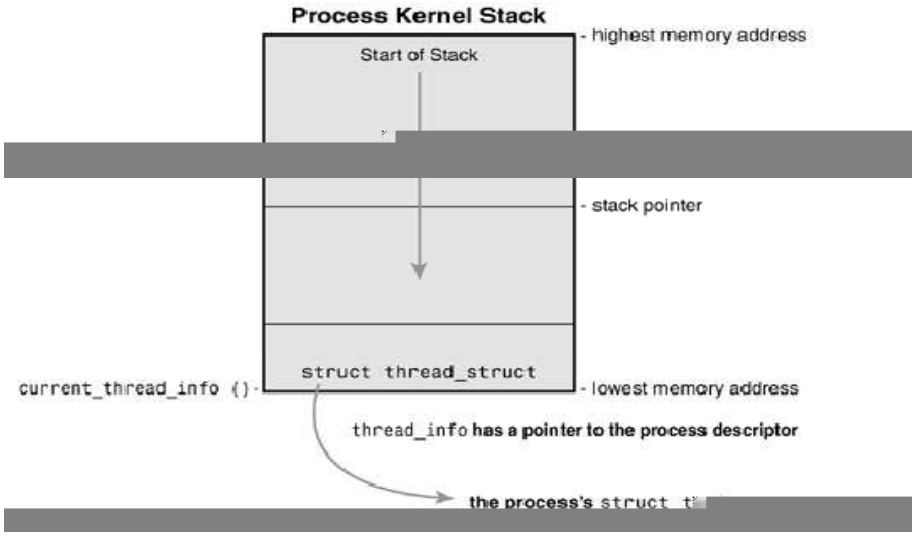


Figura 2.7: Descriptor de proceso y pila del sistema.

empleaba el puntero a la cima de la pila como registro base.

En esta nueva version de Linux, los descriptores de procesos son creados de manera dinámica utilizando el *slab allocator* ([8]). Para acceder a ellos, se ha creado una nueva estructura al final de la pila de sistema (donde estaba antes el descriptor de procesos) denominada thread_info, en la cual, entre otros campos, hay un puntero al descriptor (campo task).

En el código del planificador, es muy comun acceder al descriptor del proceso que está actualmente en ejecución, por ello existe una macro denominada current para acceder a este descriptor de manera más eficiente. Esta macro esta implementada para cada arquitectura en $include/asm - */current.h$. En algunas de ellas, como la $x86$, invoca a la función current_thread_info() (implementada en ensamblador) para obtener el thread_info del proceso, y luego su descriptor.
En la figura 2.7 ([8]) se puede observar la estructura thread_info situada en la pila de sistema del proceso.

Inicio de un Proceso

Tradicionalmente, al crear un proceso hijo con la llamada al sistema $fork()$, casi todos los recursos que posea el proceso padre eran duplicados y copiados al hijo. De manera

que sólo se diferenciaban en el PID, el PPID (parent's PID) y en algunas estadísticas y recursos tales como las señales pendientes.

Esta aproximación es bastante ineficiente, sobre todo si lo primero que ejecuta el proceso hijo es una llamada a *exec*() la cual carga un nuevo programa en el espacio de memoria, convirtiendo así, en innecesaria y en una pérdida de tiempo, la copia del espacio de memoria del proceso padre (que fácilmente puede alcanzar los 10 MB).

Actualmente, esta llamada al sistema emplea un mecanismo denominado Copy-On-Write (COW), cuyo ob jetivo es el de retrasar, o incluso evitar, la copia innecesaria de recursos a la hora de crear un proceso hijo. Con esta técnica, en lugar de duplicar todo el espacio de memoria, el proceso padre y el hijo comparten una misma y única copia. De esta forma, los datos que contiene sólo se duplicarán si uno de los procesos intenta modificar alguna de las páginas de ese espacio de memora.

Por tanto, si el proceso hijo ejecuta un *exec*() justo después del $fork()$, ninguno de los datos del espacio de memoria del padre habrán sido duplicados. La unica carga producida por $fork()$ es la copia de las tablas de páginas del padre y la creación de un nuevo descriptor de procesos para el hijo.

Sin embargo, se puede optimizar aun más si el kernel despierta y ejecuta al proceso hijo antes de continuar con el padre (en previsión de que el hijo ejecutará inmediatamente un *exec*()). Si se hace al revés, se corre el riesgo de que el padre comience a hacer modificaciones en las páginas del espacio de memoria, y que por tanto, estas se dupliquen al hijo (innecesariamente, puesto que este ultimo ejecutará otro programa). Esta acción es llevada a cabo en la *función wake_up_new_task()* y se denomina *child-run-first*.

Colas de procesos

La principal estructura de datos del planificador es la Cola de Procesos o Runqueue y se trata de la lista de los procesos ejecutables en cada procesador. Existe una cola para cada procesador y cada proceso ejecutable sólo puede estar en una cola a la vez. Está definida como *runqueue_t* en *kernel/sched.c*, aunque podría haberse hecho en *include/linux/sched.h*, pero lo que se pretende es aislar al código del planificador y

mostrar sólo una interfaz reducida al resto del kernel.

Estructura de la Cola de Procesos

La estructura de la cola de procesos es la siguiente ([8]):

```
typedef struct runqueue runqueue_t;
struct runqueue {
    spinlock_t lock;                        // Cerrojo para proteger esta cola
    unsigned long nr_running;               // Cantidad de tareas ejecutables
    unsigned long cpu_load;                 // Carga del procesador
    unsigned long long nr_switches;         // Cantidad de cambios de contextos
    unsigned long expired_timestamp;        // Tiempo desde el anterior intercambio de arrays
    unsigned long nr_uninterruptible;       // Cantidad de tareas en estado TASK_UNINTERRUPTIBLE
    unsigned long long timestamp_last_tick; // Marca de tiempo del anterior  tick del planificador
    task_t *curr;                           // Proceso que esta ejecutando
    task_t *idle;                           // Proceso ocioso de esta cola
    struct mm_struct *prev_mm;              // Mapa de memoria virtual del proceso anterior
    prio_array_t *active;                   // Array de prioridad de tareas activas
    prio_array_t *expired;                  // Array de prioridad de taras expiradas
    prio_array_t arrays[2];                 // Arrays de prioridad
    int best_expired_prio;                  // La mayor de las prioridades de las tareas expiradas
    atomic_t nr_iowait;                     // Cantidad de tareas esperando E/S
    struct sched_domain *sd;                // Dominio de planificacion de esta cola
    int active_balance;                     // Indica si la cola necesita equilibrado
    int push_cpu;                           // CPU a la que se mandan tareas durante un equilibrado
    task_t *migration_thread;               // Proceso para la migraci\'on de tareas
    struct list_head migration_queue;       // Lista de tareas a migrar durante un equilibrado
};
```

Dado que las colas de procesos son la principal estructura de datos del planificador, existen una serie de macros definidas en *kernel/sched.c* para manejarlas:

- **cpu_rq(cpu)**: devuelve un puntero a la cola de procesos del procesador cpu.
- **this_rq()**: devuelve un puntero a la runqueue del procesador actual.
- **task_rq(p)**: devuelve un puntero a la cola donde está el proceso p.

2.4.2. La función de planificación

La función principal del planificador es *schedule*(). Su propósito es seleccionar el siguiente proceso a ejecutar. Esta función es invocada cuando un proceso desea ceder el procesador (a través del método sched_yield()), o cuando este debe ser expropiado (ver).

Podemos dividir los pasos llevados a cabo por este algoritmo en cuatro secciones o partes:

Preparativos : lo primero es realizar una serie de preparativos y comprobaciones, tales como:

- Verificar que la función no fue invocada mientras el kernel está en modo atómico o con las interrupciones deshabilitadas, ya que podrá provocar interbloqueos.

- Verificar que el proceso en ejecución no es la tarea *ociosa* (aquella que se ejecuta cuando no hay otros procesos a ejecutar) y que esta no está en estado TASK_RUNNING.

- Deshabilitar la expropiación y determinar el tiempo que el proceso actual ha estado ejecutándose. Si el proceso era interactivo, dicho tiempo es reducido para evitar que este tipo de procesos, que suelen estar esperando en operaciones de E/S, pierdan su estatus de *interactivo* debido a que puntualmente han estado ejecutándose durante un largo período de tiempo.

- Si se está en una expropiación, las tareas en estado TASK_INTERRUPTIBLE y con una señal de interrupción pendiente pasan a estado TASK_RUNNING, mientras que las que están en estado TASK_UNINTERRUPTIBLE se eliminan de la cola de tareas. Esto es así, porque los procesos *interrumpibles* y con una señal pendiente, necesitan tratarla, mientras que los *ininterrumpibles* no deberían estar en la cola de procesos.

Buscar procesos candidatos : ahora se verifica si hay tareas a ejecutar. Esta acción se realiza de la siguiente manera:

- Si no hay procesos para ejecutar en la cola, se procede a un equilibrado de carga.

- Si aun así, no hay tareas, entonces se elige a la tarea ociosa (idle task). Además, se invoca a la función $wake_sleeping_dependent()$, que pertenece a la Planificación SMT 3 . Si por el contrario, había tareas para ejecutar, primero se verifica si se puede seleccionar una de ellas o deben dejarse suspendidas.

- Esto se realiza mediante la función $dependent_sleeper()$, la cual también pertenece a la Planificación SMT, y que por tanto, solo tendrán efectos si está instalado dicho tipo de planificación.

- Si en este momento, ya no quedan procesos en el Array de tareas activas, se hace un intercambio entre los dos arrays, lo cual equivale a haber hecho el bucle de recálculo de timeslices 2.4.3.

Seleccionar tarea a ejecutar : en este momento ya hay tareas a ejecutar en el array de procesos activos. Para seleccionar la siguiente tarea a ejecutar se realizan los siguiente pasos:

- Se invoca a la función *sched_find_first_bit()* para que localice de manera eficiente, en el campo bitmap[] del array, el nivel de prioridad más alto donde haya alguna tarea ejecutable. Con esto, se evita tener que recorrer el array buscando dicha tarea. Selecciona en el array, la primera tarea de la lista que corresponde al nivel de prioridad encontrado. Por ejemplo, si la función *sched_find_first_bit()* devuelve 7, entonces el proceso de mayor prioridad es el primero de la lista de tareas que se encuentra en array → queue[7].

- Si el proceso seleccionado no es de Tiempo Real, estaba suspendido y en un estado distinto a TASK_UNINTERRUPTIBLE, es decir, el valor del campo activated de su task_t es mayor que 0, entonces es muy probable que se trate de un proceso interactivo, y por tanto, se invoca a la función *recalc_task_prio()* para que actualice su sleep_avg (el tiempo medio que pasa durmiendo) y modifique su prioridad.

- Finalmente se actualiza su posición en el Array de prioridad. Todo lo referente al campo sleep_avg y las prioridades, se explica con detalle en la sección 2.4.3. En cuanto al campo activated, ver 9.

Cambio de contexto : una vez seleccionado el siguiente proceso a ejecutar, se procede al cambio de contexto entre este y el proceso anterior de la siguiente manera:

- Desactiva el flag TIF_NEED_RESCHED en el proceso que va a ser expropiado.

- Actualiza el tiempo que el proceso ha estado ejecutándose desde el último cambio de contexto o scheduler tick (interrupción del reloj del sistema).

- Actualiza las marcas de tiempo del proceso.

- Realiza el cambio de contexto (explicado en detalle en el punto 2.4.2.

- Habilita la expropiación la cual había sido desactivada durante la planificación. Si durante este tiempo, alguna expropiación había sido solicitada, el algoritmo vuelve a comenzar.

2.4.3. Política de planificación

La Política de planificación es la responsable de un aprovechamiento óptimo del procesador, ya que es la que determina qué proceso debe ejecutarse y cuándo. Además debe hacerlo satisfaciendo dos ob jetivos contrarios: baja latencia (tiempo de respuesta rápido) y un alto rendimiento (throughput).

Para cumplir estos objetivos se diferencia entre dos tipos de procesos: los I/O-bound y los CPU-bound.
Los primeros son aquellos que pasan la mayor parte del tiempo suspendidos esperando a que finalicen operaciones de E/S, mientras que los segundos están continuamente utilizando los recursos del procesador.

Los procesos I/O-bound solo se pueden ejecutar durante periodos de tiempo cortos, ya que terminarán por suspenderse en alguna operación de E/S. Por el contrario, los CPU-bound pueden estar en ejecución hasta que sean expropiados.

Por tanto, para favorecer la baja latencia, se otorga mayor prioridad a los procesos interactivos (I/O-bound) para que estos puedan ejecutarse un mayor numero de veces. De esta manera, pueden comenzar cuanto antes sus peticiones de E/S y dejar al procesador *libre* para los procesos que más lo necesitan: los CPU-bound.

El cálculo de prioridades y de tiempos de ejecución (timeslices) se explica con detalle en las siguientes subsecciones.

Prioridades de los procesos

Linux emplea una planificación basada en prioridades la cual consiste en clasificar a todos los procesos segun su valor y necesidades de tiempo en el procesador. De esta

manera, los procesos con mayor prioridad siempre se ejecutarán antes que los de menor prioridad. Si existen varios con una misma prioridad, estos se irán seleccionando por turnos rotatorios (round-robin).
Las prioridades en Linux se encuentran representadas internamente por valores pertenecientes al rango [0..MAX_PRIO-1]. Por defecto la constante MAX_PRIO (definida en *include/linux/sched.h*) vale 140 y este es por tanto, el numero de niveles de prioridad que existen en el sistema. Cuanto menor sea este nivel para un proceso, mayor será su prioridad.

Estos niveles se encuentran divididos en dos subrangos. El primero de ellos es para procesos de Tiempo Real y comprende a los primeros MAX_RT_PRIO niveles, es decir, el rango [0..MAX_RT_PRIO-1], donde la constante MAX_RT_PRIO (también definida en *include/linux/sched.h*) vale por defecto 100. Todo lo referente a los procesos de Tiempo Real se explica en la sección 2.4.3.

El segundo subrango ([MAX_RT_PRIO..MAX_PRIO-1]) que es el que se tratará en esta subsección, incluye a los 40 niveles restantes. En estos niveles estarán la mayoría de los procesos de usuario.

Prioridades Estáticas

Cuando se crea un proceso de usuario, se le asocia una prioridad base igual a la del proceso padre. Esta prioridad es denominada Prioridad Estaática, ya que es la que tendrá este nuevo proceso a lo largo de su vida. El nivel de prioridad al que pertenecen estos procesos es almacenado en el campo static_prio de la estructura task_t (ver 2.4.1).

Es posible modificar la prioridad estática de un proceso de usuario mediante la llamada al sistema *nice*(). Al invocarla, se ejecuta en *kernel/sched.c* el método *sys_nice()*, el cual, aparte de algunas comprobaciones, llama a *set_user_nice()*. Esta función establece la nueva Prioridad Estática del proceso y actualiza su posición en el Array de Prioridades.
Sin embargo, en esta llamada al sistema, el valor de la nueva prioridad introducido como parámetro, no es uno de los 40 niveles de prioridad en el que se ejecutan los procesos de usuario, sino una conversión al rango [-19..0..20].
Los valores de este nuevo rango se denominan también nice. De esta manera, los procesos con mayor prioridad serán aquellos que tengan un nice negativo, mientras que los positivos tendrán menor prioridad (están siendo *amables* con el resto de procesos al

reducir su prioridad).
Se puede consultar el nice de un proceso de usuario, a través de las funciones *task_nice()* y *task_prio()* en *kernel/sched.c*.

Prioridades Dinámicas y Prioridad Efectiva

Tal y como se mencionó al inicio de esta sección, el planificador de Linux siempre favorece a los procesos interactivos y penaliza a los intensivos en cálculo.
Para llevar a cabo esta acción, **modifica ligeramente las prioridades estáticas** de los procesos, aumentando la de los primeros (disminuyendo su nivel de prioridad) y reduciendo la de los segundos (incrementando su nivel).
En concreto, este a juste en sus prioridades consiste en *bonificar* o penalizar a los procesos con hasta 5 niveles de prioridad. A estas bonificaciones y penalizaciones se les denomina Prioridades Dinámicas, ya que el a juste se realiza de manera continua y siguiendo el comportamiento de cada proceso.

El nuevo valor de la prioridad queda almacenado en el **campo prio de la estructura task_t (ver 2.4.1) de cada proceso**, ya que este es el campo que realmente indica la prioridad de todos los procesos (ya sean de usuario o de tiempo real) y el utilizado como índice a la hora de insertarlo en los Arrays de Prioridad.

Para saber qué procesos deben ser penalizados o beneficiados en sus prioridades, el planificador emplea una heurística basada en el tiempo medio que cada proceso pasa suspendido (muy probablemente en operaciones de E/S). Así, cuanto mayor sea la media de un proceso, mayor será la reducción en sus niveles de prioridad (hasta un límite de 5 niveles por deba jo de su prioridad estática). Este tiempo medio se guarda en cada proceso en el campo sleep_avg de la estructura task_t (ver 2.4.1).

Cuando un proceso se despierta, el tiempo que ha pasado suspendido pasa a formar parte de su media. Esta acción es realizada en la función *recalc_task_prio()*, la cual es invocada desde *activate_task()*. También es invocada desde *schedule*() cuando el proceso seleccionado para ejecución estaba suspendido, ya que además se contabiliza el tiempo que pasa en la runqueue desde que es despertado hasta esa primera selección.

Sin embargo, esta función no incrementa el valor de la media si el proceso estaba suspendido en estado TASK_UNINTERRUPTIBLE (el campo activated de su estructura task_t vale -1), ya que podrá tratarse de un CPU-bound que ocasionalmente estaba suspendido en una operación de E/S. Finalmente, invoca a effective_prio() para que actualice su prioridad y la almacene en el campo prio de su estructura task_t.

La función effective_prio() es la calcula el *bonus* o la penalización en las prioridades. Además de ser invocada desde recalc_task_prio(), también lo es desde scheduler_tick(). En cualquier caso, es siempre después de haber modificado el sleep_avg de un proceso, ya que esto es la base de la heurística. Básicamente lo que realiza es escalar el tiempo medio que un proceso pasa suspendido ([0..MAX_SLEEP_AVG]) al rango de bonus/penalizaciones [-5..0..+5]. La constante MAX_SLEEP_AVG (definida en *kernel/sched.c*), cuyo valor por defecto es 10ms, indica el valor máximo que puede alcanzar sleep_avg.

A pesar de esta modificación en la prioridad, como sólo afecta al 25 % del los 40 niveles de prioridad pertenecientes a los procesos que no son de Tiempo Real, se consigue *respetar* la elección del usuario del nice de los procesos. Por ejemplo, un proceso interactivo con nice +19 (nivel de prioridad 139) nunca podrá expropiar a un proceso intensivo en cálculo que tenga un nice 0 (nivel 120), o bien, un proceso CPU-bound de nice -20 (nivel 100), nunca será expropiado por uno interactivo de nice 0.

Equidad al crear y destruir procesos

Un proceso recién creado siempre recibe como nice inicial, la prioridad estática de su proceso padre. Además, la función *wake_up_new_task()* reduce el *sleep_avg* tanto del proceso padre como del hijo recién creado. Esto impide que un proceso interactivo que tenga una media alta genere muchos procesos hijos, también con una media alta, y terminen monopolizando al procesador.
Por otro lado, cuando una tarea muere, si su media es menor que la de su proceso padre, entonces también se reduce la media de este último. Esto se realiza en la función *sched_exit()*.

Timeslices

El timeslice de un proceso es el tiempo que este puede estar ejecutándose en el procesador antes de ser expropiado. En general, y para cualquier Sistema Operativo, el

determinar un valor por defecto no es una tarea sencilla, ya que uno muy largo reduce la interactividad del sistema e incluso produce la inanición del resto de procesos, mientras que uno muy corto reduce significativamente el rendimiento debido a la sobrecarga del planificador y los cambios de contexto. Por otro lado, los procesos CPU-bound necesitan timeslices largos (por ejemplo, para mantener sus datos en la cache), mientras que a los I/O-bound les basta con uno corto.
Para que el planificador del Linux favorezca a los procesos interactivos, sin penalizar excesivamente a los intensivos en cálculo, se ha optado por darles más prioridad a los primeros, pero otorgando a todos un timeslice por defecto (es decir, para el nivel de prioridad 120) relativamente alto (100ms).

En cualquier caso, el timeslice de un proceso es calculado unicamente en función de su Prioridad Estática mediante la función *task_timeslice()*.

Tipo de Tarea	**Nice**	**Timeslice**
Recién creada	El del padre	La mitad de la del padre
Miníma prioridad	+19	$5ms$ (MIN_TIMESLICE)
Prioridad por defecto	0	$100ms$ (DEF_TIMESLICE)
Máxima prioridad	−19	$800ms$ (DEF_TIMESLICE)

Cuadro 2.1: Asignación de los timeslices del Planificador

Cálculo del timeslice

La función *task_timeslice()* convierte el rango de las prioridades de los procesos de usuario ([MAX_RT_PRIO..MAX_PRIO-1]), al rango de tiempos [800ms..100ms..5ms] basándose en el campo *static_prio* (prioridad estática) de los procesos. Cuanto menor sea este valor para un proceso, mayor será su tiempo de ejecución. Sin embargo, cualquier proceso, por muy poca prioridad (nivel alto) que tenga, recibirá un valor superior o igual a MIN_TIMESLICE (5ms).
Existe una llamada al sistema para obtener el timeslice de un proceso. Se trata de *sched_rr_get_interval()* (método *sys_sched_rr_get_interval()*, definido en *kernel/sched.c*), aunque aparte de algunas comprobaciones, es una simple llamada a *task_timeslice()*.
Podemos resumir los timeslices que reciben los procesos segun su prioridad estática en la tabla 2.4.3.

Actualizando el timeslice de un proceso

El timeslice de los procesos se guarda en el campo *time_slice* de la estructura task_t (ver 2.4.1) y se actualiza en la función *scheduler_tick()*, la cual es llamada regularmente a través de una interrupción del reloj.
Normalmente, cuando un proceso consume completamente su tiempo de ejecución se le otorga un nuevo timeslice, se calcula su prioridad con la función *effective_prio()* y se mueve del array de procesos activos al de expirados.
Además se invoca a *set_tsk_need_resched()* para que el planificador lo expropie y seleccione a otra tarea para su ejecución.
Sin embargo, para impedir que los procesos interactivos se queden mucho tiempo en el Array de expirados, esperando a que todos los demás procesos de menor prioridad se suspendan o consuman sus respectivos timeslices (sobre todo los CPU-bound, que se ejecutarán hasta que sean expropiados), estos primeros son reinsertados en el Array de Activos (al final de la lista de procesos de su nivel de prioridad). Esto no significa que vuelven a entrar en ejecución inmediatamente, sino que en un futuro serán seleccionados por el planificador. La interactividad del proceso es determinada con la macro *TASK_INTERACTIVE()*.

No obstante, para prevenir la inanición de los procesos en el array de expirados, se consulta a la macro *EXPIRED_STARVING()* para saber si ha transcurrido mucho tiempo desde el ultimo intercambio de arrays, en cuyo caso el proceso interactivo no es reinsertado en el Array de activos sino en el de expirados.
Finalmente, también se evita que los procesos interactivos con un timeslice muy alto puedan consumirlo completamente en un solo turno de ejecución, monopolizando así el procesador. En estos casos se expropian y también son reinsertados en el Array de activos, al final de la lista de procesos de su nivel de prioridad. Su timeslice no se pierde sino que es como si se dividiese en trozos más pequenos. Esto se controla con la macro *TIMESLICE_GRANULARITY()*.

Equidad al crear y destruir a procesos

Al crear un proceso, el timeslice sin consumir del proceso padre se divide a partes iguales entre este y el hijo, de manera que el tiempo de ejecución o total no cambia. Esto se realiza en la función *sched_fork()*.
De la misma manera, cuando un proceso hijo muere, su timeslice sobrante es retribuido al padre, recuperando así, parte del tiempo de ejecución que perdió al crearlo. Esta acción se lleva a cabo en el método *sched_exit()*.

Expropiación

Una de las nuevas características del Linux 2.6 es que es completamente expropiativo, es decir, puede expropiar a un proceso independientemente del modo en el que se encuentre el S.O. (usuario o nucleo). Esto ha obligado a cambiar la forma de gestionar los recursos del kernel, en especial las estructuras de datos las cuales fueron dotadas de mecanismos de bloqueo (locks) para controlar los accesos concurrentes por parte de varios procesos (ver, por ejemplo, las runqueues en la sección 2.4.1).

Cómo y cuándo invocar al planificador

El kernel debe saber cuándo se debe llamar a la función *schedule()* sin esperar a que un proceso lo haga explícitamente, ya que entonces este ultimo podría ejecutarse indefinidamente. Para ello, existe un flag denominado *TIF_NEED_RESCHED*, el cual está definido en *include/asm-*/thread_info.h* y se almacena en la estructura *thread_info* de cada proceso. Dicho flag es consultado en distintas ocasiones por el nucleo, de manera que cuando este último detecta a un proceso que lo tiene activado, invoca a *schedule()* para que seleccione a una nueva tarea (los instantes en que se consulta al flag y se procede a la expropiación, se explican con detalle más adelante, en siguientes subsecciones).

El flag es activado en el método *scheduler_tick()* cuando un proceso consume todo su timeslice, en *try_to_wake_up()* cuando se despierta a un proceso con mayor prioridad que el que está en ejecución, en *wake_up_new_task()* por la misma razón pero al crear un nuevo proceso, y también durante el equilibrado de carga (ya que un proceso que es migrado a otro procesador puede tener mayor prioridad que el que se está ejecutando en ese procesador destino. En cualquier caso, es el planificador (en el método *schedule()*) el que lo desactiva cuando ya ha seleccionado una nueva tarea a ejecutar.

Planificación en Tiempo Real

El planificador de Linux también proporciona una planificación en tiempo real suave (soft real-time), es decir, una planificación en la que se intenta cumplir los plazos de ejecución de los procesos (dead line) pero sin tener una garantía de ello.

Tal y como se mencionó al explicar los Niveles de Prioridad en Linux (2.4.3), el rango de prioridades de los procesos de tiempo real es el comprendido entre 0 y MAX_RT_PRIO-

1, y que por defecto son los primeros cien niveles. Por tanto, este tipo de procesos siempre expropiarán a a los de usuario.

Esquemas de planificación

Linux emplea diferentes esquemas o políticas de planificación dependiendo del tipo de proceso. Para procesos que no son de tiempo real (los de usuario), se emplea SCHED_NORMAL, que es el esquema por defecto. Por el contrario, para procesos de tiempo real se dispone de los siguientes esquemas:

- SCHED_FIFO: es una planificación de tipo FIFO (first-in-first-out) en la que no existen timeslices, sino que cada proceso se ejecuta hasta que se suspende o explícitamente cede el procesador. Cuando existe más de un proceso SCHED_FIFO, se elige al de mayor prioridad (menor nivel).

- SCHED_RR: es como SCHED_FIFO salvo que sí existen timeslices y por tanto, cuando un proceso lo consume, este pasa al final de la cola de su nivel de prioridad y se elige al siguiente para su ejecución (planificación tipo round-robin). Procesos con este esquema de planificación tienen menos prioridad que los SCHED_FIFO y por tanto pueden ser expropiados por estos últimos.

El esquema de planificación de cada proceso se almacena en el campo policy de la estructura task_t (ver 2.4.1). Cuando un proceso tiene un esquema de planificación de tiempo real, el planificador no emplea Prioridades Dinámicas(ver 2.4.3), sino que opera únicamente con el valor del campo prio (sin modificarlo) e ignora el de *static_prio* (de hecho, la llamada al sistema nice() no tiene ningun efecto sobre este tipo de procesos). Con esto se asegura que se respetará de forma estricta su nivel de prioridad. Cambiando las prioridades de los procesos de Tiempo Real Linux proporciona una serie de Llamadas al Sistema para poder modificar y consultar la prioridad y el esquema de planificación de los procesos de tiempo real (nice() sólo afecta a los procesos de usuario). Estos métodos se resumen en la tabla 2.4.3.

Al igual que ocurre con nice(), utilizada para los procesos de usuario, en estas Llamadas al Sistema los valores de prioridad devueltos o introducidos como parámetros no son los internos del planificador (los Niveles de Prioridad), sino una conversión al rango [1..MAX_USER_RT_PRIO-1], donde 1 indica un proceso con la menor prioridad y MAX_USER_RT_PRIO-1 el de mayor. El resultado de dicha conversión se almacena

nice()	Le da a un proceso el valor de nice indicado
sched_setscheduler()	Modifica la politica de planificación de un proceso
sched_getscheduler()	Consulta la politica de planificación de un proceso
sched_setparam()	Configura la prioridad de tiempo real
sched_getparam()	Obtiene la prioridad de tiempo real
sched_get_priority_max()	Consulta la máxima prioridad de tiempo real
sched_get_priority_mim()	Consulta la mínima prioridad de tiempo real
sched_rr_get_interval()	Consulta el valor del timeslice de un proceso
sched_setaffinity()	Consulta el valor del timeslice de un proceso
sched_setaffinity()	Consulta el valor del timeslice de un proceso

Cuadro 2.2: Algunas llamadas al sistema relativas a la prioridad de los procesos de Tiempo Real

en el campo *rt_priority* de la estructura task_t (ver 2.4.1) de cada proceso y sólo vale 0 si ese proceso se ejecuta con el esquema de planificación SCHED_NORMAL.

MAX_USER_RT_PRIO es una constante definida en *include/linux/sched.h* y por defecto vale lo mismo que MAX_RT_PRIO (100). Sin embargo, si la primera tiene un valor menor, permite reservar los primeros niveles de prioridad a los kernel threads (procesos que sólo se ejecutan en modo kernel). Para más detalles sobre estas llamadas al sistema, se puede consultar las páginas del Manual de Programación de Linux.

2.5. Mecanismos para la depuración en el kernel

- **printk()**: En parte del código se ha introducido la función printk, la cual es muy similar a la conocida printf sólo que opera dentro del kernel.
  Esta función en su llamada puede acompañarse de un loglevel, el cual permite decidir al kernel si debe imprimir el mensaje por consola o basta con escribirlo en el log del sistema. El que más usaremos será el loglevel ¡1¿que indica la prioridad del mensaje; se ha especificado una alta prioridad (bajo número) para que el mensaje aparezca por pantalla y no se quede en los ficheros de mensajes del kernel.

- **Comunicación interactiva mediante** */proc*: Como se verá en el apartado 6 es interesante tener una zona de memoria a través de la cual se pueda obtener información sobre el estado del algoritmo de Calidad de Servicio o de las estadísticas que se están midiendo.
  Como alternativa a la creación de nuevas llamadas al sistema se pueden introducir nuevas entradas en el sistemas de archivos virtual */proc* configurando cómo se dsea leer y cómo se desea escribir.
  Cabe comentar que hay que realizar estas operaciones con sumo cuidado para no provocar desbordamientos debido al limite de tamaño del buffer de transferencia.

- **Introducción de llamadas al sistema**: Las llamadas al sistema ofrecen una interfaz mediante la cual los procesos corriendo en el espacio de usuario pueden interactuar co el sistema. Esta interfaz le da a las aplicaciones acceso al hardware y a otros recursos del sistema operativo. Las llamadas al sistema actuan como *mensajeros* entre las aplicaciones y el kernel. El hecho de que este tipo de interfaz exista, y de que las aplicaciones no tengan un acceso directo a cualquier información del sistema, es clave para proporcionar una estabilidad al sistema y evitar errores fatales.
  No obstante, Linux ofrece menos llamadas al sistema que la mayoría de los sistemas operativos.Incluir una llamada al sistema nueva es una tarea sencilla pero poco recomendable. En la mayoria de situaciones en las que se podria resolver un conflicto mediante una llamda al sistema suele haber más de una alternativa distinta para resolver ese problema.
  Entre varios de los contras de crear una llamada al sistema nueva están que se necesita asignar un número de llamada al sistema, que necesita haber sido asignado oficialmente durante una tarea de desarrollo del kernel para conseguir concordancia con las llamadas que otros desarrolladores hayan introducido y que es necesario adaptar a cada arquitectura dicha llamada al sistema y además, la arquitectura que que poder soportarla.

2.6. Aplicaciones existentes para monitorización mediante $MSR'S$

La información sumistrada por los contadores hardware puede ser usada para modificar el sistema optimizando sensiblemente su rendimiento.
Es por ello que existen diversas aplicaciones hoy en día que pemiten configurar estos contadores en una determinada máquina o simplemente dar al usuario la configuración apropiada que deberían tener para que detectasen un determinado evento.
A continuación se dará un breve repaso a las aplicaciones existentes para monitorizar eventos mediante contadores hardware junto con los motivos por los cuales se descartaron como ayuda para el estudio del rendimiento en el algortimo de calidad de servicio.
Se hará una breve reseña además a dos aplicaciones interesantes relacionadas con el servicio de tareas de tiempo real: el *parche smt* existente para linux y *RTLinux*.

2.6.1. *Brink and Abyss Pentium 4 Performance Counter Tools For Linux*

La herramienta *Brink* en un script en perl que proporciona una interfaz para los contadores de rendimiento de la familia de procesadores Pentium 4 en un sistema Linux.
Los archivos de configuración de experimentos con los que trabaja la herramienta describen una colección de programas para ser ejecutados y un conjunto de eventos para ser monitorizados. Para cada programa especificado y cada conjunto de eventos, *Brink* configura el procesador para monitorizarlos, ejecuta el programa y cuando este termina, proporciona varios archivos de salida con la información más relevante.
Inicialmente, es necesario suministrar al programa un fichero .xml en el que se agrupan los eventos que se desean monitorizar simultáneamente agrupados como experimentos.

Brink permite medir tanto eventos precisos (*precise event-based sampling*) como no precisos.
El manejo de esta herramienta es complejo: para usar *Brink* es necesario un profundo conocimiento de los contadores de rendimiento del Pentium 4 como dice en la descripción accesible en la web oficial de *Brink* [12].
Brink proporciona dos ficheros de configuración: *Experiment configuration file* y *EMON configuration file*. *Experiment configuration file* describe la configuración para poner en marcha los contadores usando los nombres de varios campos MSR más que posiciones a nivel de bit. Además, *Brink* toma decisiones por el usuarios como la combinación ESCR/CCCR/Contador que se debe tomar de las disponibles.

Si varios eventos se incluyen en el fichero de entrada como parte de un mismo experimento se chequean dependencias entre los mismos. De haberlas, se muestra solo un error en el fichero de salida desconsiderando el resto de eventos que no estuviesen implicados en la incompatibilidad.
El *EMON configuration file* describe los detalles de los contadores incluyendo los registros específicos MSR para detectar los eventos, contarlos y los registros con los que se debe hacer el marcaje de ser necesario.

Abyss es un programa a nivel usuario hecho en C que actua como interfaz con un módulo especial que permite controlar la puesta en marcha de los contadores y el conteo de los eventos.
Para más información re recomienda consultar la página oficial [12].

2.6.2. *Perfctr*

Se trata de un parche para Linux que también permite medir con exactitud los eventos que están ocurriendo en un momento dado.
Facilita funciones para leer los contadores de rendimiento hardware pero al contrario que otras es dependiente de la arquitectura del sistema. Además su uso es más complejo para proponerlo como una herramienta de análisis del rendimiento para un usuario cualquiera.

2.6.3. *Papi: Performance Application Programming Interface*

Esta es una herramienta multiplataforma que proporciona dos interfaces para el análisis de rendimiento.
Una a muy bajo nivel que programa los contadores para obtener mediciones de conjuntos de eventos. La otra, a más alto nivel está dedicada a proporcionar los datos al usuario mediante gráficas ofreciendo la posibilidad de parar, reiniciar o leer un evento específico.
No es apropiada para su uso como herramienta de profiler en el algortimo de calidad de servicio por el tipo de interacción que tiene con el sistema. Además la configuración de los eventos de alto nivel que ofrece es poco flexible y necesitamos una herramienta más versatil y potente.
Para más información se recomienda consultar la página oficial [2].

Parte II

Implementación

Capítulo 3

Brank

3.1. Introducción

3.1.1. Problemática introducida por el sistema de monitorización de eventos

El sistema hardware de monitorización de eventos, es dependiente de la arquitectura. En los procesadores con SMT de Intel (tecnología *Hyperthreading*), la configuración necesaria para la medida de un evento hardware requiere al menos de la configuración de tres registros específicos: un contador, un registro de control para el contador y un registro detector de eventos (como se describe en la sección pertinente). Estos registros están especializados en la detección, cuenta o control de unos eventos determinados. Además de la especialización de cada registro, las conexiones entre ellos están muy limitadas, pudiendo usar conjuntamente un número muy reducido de ellos. Por otra parte cabe destacar que existen 44 detectores de eventos, 18 contadores y 18 registros de control de contador.

Esta implementación del sistema de monitorización de eventos en los procesadores de Intel con tecnología *Hyperthreading* tiene algunos inconvenientes:

- Los recursos hardware destinados a monitorización son limitados
- Los eventos hardware son de muy bajo nivel expresivo
- La configuración de los registros específicos es compleja
- No todos los eventos pueden monitorizarse de manera independiente en cada procesador lógico

El hecho de que los recursos hardware de monitorización estén limitados, origina que surjan incompatibilidades entre conjuntos de eventos. Un conjunto de eventos hardware es incompatible si no hay recursos de monitorización disponibles para realizar la cuenta de todos los eventos del conjunto simultáneamente.

Por lo general, en la implementación de Intel, un evento hardware tiene asociado dos detectores de eventos para realizar su monitorización. El problema radica en que un detector de eventos puede ser configurado para monitorizar varios eventos –mediante distintas configuraciones–, y sólo detecta un evento a la vez. Por lo tanto, una conjunto de eventos que contenga tres eventos hardware con dos detectores de eventos asociados a los tres, será incompatible. Uno de los eventos del conjunto no podrá monitorizarse.

Como indicamos anteriormente, los eventos hardware realizan monitorizaciones de bajo nivel. Esto provoca que sea necesaria la cuenta de más de un evento para obtener medidas que representen parámetros de rendimiento. Por ejemplo, el sistema de monitorización no permite la cuenta directa de un evento que mida la tasa de fallos de primer nivel de cache. Para obtener esta tasa, hemos de recurrir a dos eventos hardware: el número de fallos de cache de primer nivel y el número de instrucciones de carga (`LOADS`) retirados. La misma situación ocurre para medidas como el IPC (instrucciones por ciclo).

Para realizar la configuración de la monitorización de eventos, es necesario proceder a la escritura de números de 32 bits en dos registros –el detector de eventos (o ESCR) y el registro de control de contador asociado (CCCR)– así como realizar el borrado del contador donde se almacena el resultado de la cuenta. Estas secuencias binarias de 32 bits están divididas en subsecuencias se que corresponden con campos de los registros de configuración[1]. La información de configuración de algunos campos de los registros, es específica a esa instancia de registro concreto; aspecto que provoca que la configuración de los eventos sea una tarea tediosa causada por esta heterogeneidad.

La documentación que proporciona Intel, para gestionar el sistema de seguimiento de eventos hardware, da una idea de la complejidad para realizar esta tarea. En el documento de programación de sistemas (falta referencia) existen tablas de considerable longitud, que *explican* el funcionamiento del sistema.

Por último, resulta necesario destacar una característica inherente a la implementación de Intel, que limita en cierta manera el seguimiento independiente de procesos

[1]Esta división en subsecuencias de bits es similar a la que aparece en las secuencias de bits que representan instrucciones de una arquitectura hardware, donde existen varios campos: código de operación, operandos, flags de suboperación,...

que ejecutan en distintos procesadores lógicos. La mayor parte de los eventos hardware disponibles, permiten detectar situaciones independientes en cada procesador lógico; es decir, un evento que ocurre en un procesador lógico solo incrementa el contador ocurrencias de ese evento si la configuración de monitorización así lo indica. En cambio, existen otros eventos –denominados *TI (Thread Independent)* en la terminología de Intel– en los que el contador de ocurrencias de ese evento se incrementa de manera global, o lo que es lo mismo, cada vez que el evento se produzca en uno de los dos procesadores lógicos.

3.1.2. Soluciones aportadas por *Brink*

Brink, es un programa desarrollado por B. Sprunt que proporciona una interfaz de alto nivel para los mecanismos de monitorización del rendimiento del Intel Pentium IV. Se trata de un programa de línea de comandos, que recibe la entrada en un fichero XML, donde se le especifica un conjunto de eventos hardware con distintas características configurables. Brink genera, a partir de esta configuración en XML, un fichero de texto donde se indican los registros específicos a utilizar –para llevar a cabo dicha monitorización– junto con las secuencias binarias de 32 bits, asociadas a la configuración de cada uno de ellos.

Brink, aporta una solución a la complejidad de configuración del sistema de monitorización de Intel. Con él, ya no es necesario recurrir a las *"tablas informativas"* de la documentación de Intel; sino que basta con escribir un fichero XML y capturar la configuración hardware que proporciona el programa como salida de su procesamiento.

Además de la simplicidad de la configuración, *Brink* chequea las inconsistencias entre conjuntos de eventos hardware. Si la configuración de entrada constituye un conjunto de eventos incompatibles, está situación es detectada por el programa y mostrada al usuario junto con la causa que provoca la inconsistencia.

Cabe destacar que *Brink* tiene otras muchas facilidades para dar soporte a la gestión de la monitorización; sin embargo, hemos destacado estas dos características que resuelven dos de los problemas esenciales unidos a la gestión de la monitorización del rendimiento. Para más información visite el sitio web del autor.

3.1.3. Motivación de *Brank*

Brink es una herramienta para Linux escrita en Perl que constituye una buena solución para gestionar la configuración de eventos hardware. Sin embargo, también tiene algunas limitaciones:

- No es multiplataforma
- No existe ningún *frontend* –o aplicación con interfaz gráfica de usuario basado en *Brink*– que facilite la creación de ficheros de configuración XML para *Brink*
- Solo tiene soporte para eventos hardware específicos de Intel, no ofreciendo ninguna abstracción de más alto nivel

Con el objetivo de superar todas estas limitaciones surge *Brank*, una herramienta gráfica multiplataforma para gestión del sistema de monitorización del rendimiento para Intel Pentium IV.

Brank ofrece una GUI (*Graphical User Interface*), para *Brink*. Los ficheros XML que sirven de entrada a *Brink* son generados de manera automática por *Brank* y, de este modo, la comunicación entre ambas aplicaciones se realiza de manera transparente al usuario. El fichero de texto, que constituye la salida de *Brink*, es analizado por un *parser* específico, cuya función es almacenar la información –obtenida a través de dicho análisis– en estructuras de datos internas. Una vez capturada la información de configuración hardware, es mostrada al usuario de manera estructurada en la interfaz gráfica.

Figura 3.1: Integración entre *Brink* y *Brank*

Por otro lado, *Brank* dispone de un soporte para medición de eventos de más alto nivel que los básicos eventos hardware. Para ello se han diseñado unas estructuras de datos y representaciones específicas a dos niveles: nivel de usuario y nivel de *kernel*. Las estructuras utilizadas a nivel de usuario se especifican en la sección 3.5.

3.2. Requisitos de *Brank*

3.2.1. Multiplataforma

Una de las características deseables, para una aplicación que genera configuraciones para hardware específico de una arquitectura –como *Brank*–; es el hecho de que esté disponible una implementación para todos los sistemas operativos existentes para esta

arquitectura. En el caso de Intel, merece la pena citar los sistemas operativos de Linux , Apple Mac OS X (en desarrollo para Intel) y Microsoft Windows.

Partiendo de la base de que *Brank* utiliza *Brink* – para obtener la información básica sobre eventos hardware– y de que *Brink* es una aplicación específica para Linux –por tanto, no es multiplataforma–; hemos tenido que portar *Brink* a Microsoft Windows, para conseguir que *Brank* sea multiplataforma.

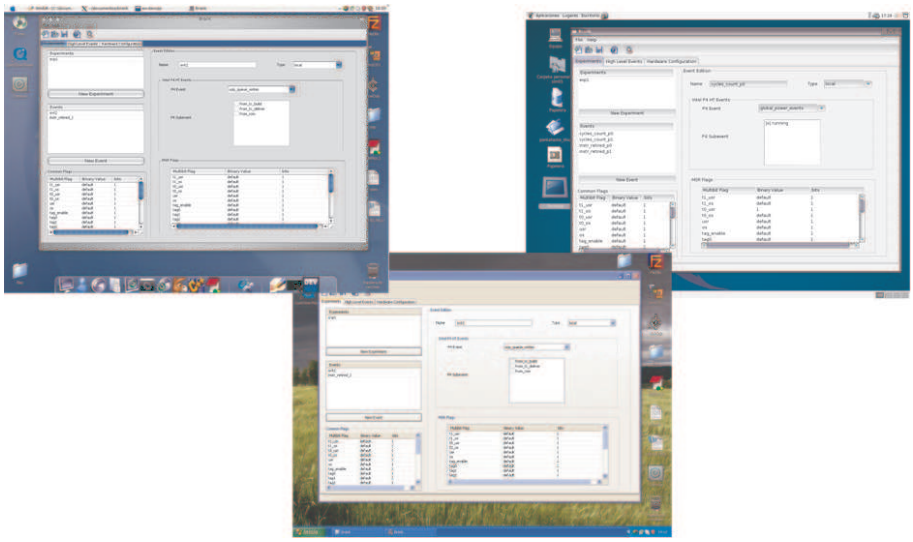


Figura 3.2: Apariencia de *Brank* en distintas sistemas operativos: Mac, Linux y Windows

Para lograr una aplicación, más eficiente y más integrada con el sistema operativo, se ha escogido `C++` como lenguaje de implementación para *Brank*. El hecho de haber escogido `Java`, hubiera supuesto más sencillez a la hora de lograr la portabilidad; pero hubiera resultado más compleja la tarea de integración con una aplicación en `Perl` -como *Brink*-. Incluso, con `Java`, hubiéramos conseguido una solución menos eficiente.

Como la librería estándar de `C++` carece de una implementación independiente del sistema operativo, para el desarrollo de aplicaciones con interfaz gráfica de usuario; hemos escogido *wxWindows* –librería multiplataforma– para implementar la interfaz gráfica de *Brank*.

3.2.2. Herramienta de apoyo para el planificador *simbiótico*

El objetivo esencial de nuestro proyecto, es es desarrollo de un planificador *simbiótico* para el kernel de Linux, con soporte para calidad de servicio en procesadores con

SMT de Intel. Para conseguir, la calidad de servicio –como comentamos en secciones previas– se detectan situaciones no deseadas en las que un proceso prioritario compite por recursos compartidos del procesador. Esto se realiza mediante la monitorización de eventos hardware o combinaciones de ellos.

Para introducir experimentos de monitorización con control de umbrales de violación de la calidad de servicio[2], es preciso disponer de un mecanismo automatizado para generar las estructuras de datos del kernel mediante las cuales se representan estos experimentos.

Brank es un programa de usuario que facilita –entre otras cosas– la generación de ficheros de configuración XML que servirán como entrada al *kernel* de Linux; ficheros que describen experimentos de monitorización. Para la generación de estos ficheros, se realiza un cambio de representación para los conjuntos de eventos, entre la utilizada en *Brank* y una representación en XML muy similar a la utilizada en el núcleo de Linux. Esta tarea supone la conversión entre representaciones de muy distinto nivel de abstracción.

La representación usada por *Brank* para describir los conjuntos de eventos, se realiza por medio de la utilización de unas estructuras de datos especializadas en la inserción, actualización y borrado de eventos del conjunto individualmente. Esta representación, difiere de la que se usa en el núcleo donde se persiguen otros fines como la optimización de los algoritmos de cuenta de eventos.

3.2.3. Basado en XML

El lenguaje de marcado extensible (XML), es un formato de texto plano, que permite, entre otras cosas, la representación de información estructurada de manera muy compacta. Por ser un formato de texto, presenta la ventaja de ser legible y modificable por los humanos sin necesidad de utilizar herramientas adicionales (a diferencia de los ficheros binarios). Además, para realizar su procesamiento para la extracción de información desde una aplicación, no es preciso implementar un *parser* "ad hoc"; sino que, por el contrario, existen API's que ofrecen analizadores genéricos y mecanismos muy avanzados de validación de este tipo de documentos.

[2]Un umbral de violación de calidad de servicio, es un valor entero positivo que constituye el máximo (o mínimo) valor que puede tomar un parámetro de control de Qos (como el IPC, los fallos de predicción de saltos,...), que indican una situación no deseada en la que se produce una violación del trato preferente dado a un proceso

Por otra parte *Brink*, recibe la entrada en formato XML, por lo que la generación automática de XML por parte de *Brank* es obligada.

Por todo esto, el procesamiento XML es una pieza clave para *Brank*. La aplicación exporta información en XML para el núcleo de Linux, sus ficheros de configuración están en XML, los archivos de *Brank* que representan colecciones de eventos también están en este formato,...

3.3. Uso de *Brank*

En esta sección se describen aspectos relacionados con la funcionalidad de *Brank*. En una primera instancia, se abordará la descripción de las tareas esenciales de *Brank*. Posteriormente, nos centraremos en la descripción de la interfaz gráfica de la aplicación, mediante un recorrido por sus distintas secciones.

3.3.1. Funcionalidades de *Brank*

Brank maneja experimentos de monitorización. Los ficheros de *Brank*, en formato XML, almacenan conjuntos de experimentos de monitorización y la aplicación es capaz de abrirlos y guardarlos. Cada experimento de monitorización está constituido por un conjunto de eventos de alto y bajo nivel.

Los eventos de bajo nivel son los eventos hardware tal como los concibe *Brink*. Todas las características y parámetros que los describen, tales como el tipo de evento hardware, la máscara de subevento, flags de ESCR y CCCR ...; pueden configurarse mediante la interfaz gráfica. Los conjuntos de eventos de bajo nivel son los que se agruparán con todos sus parámetros en un fichero XML generado automáticamente por *Brank*. Cuando se pulsa el botón para obtener la configuración de hardware, este fichero XML constituirá la entrada de *Brink*. Una vez ejecutado el procesamiento se procede al análisis del *fichero de LOG* de *Brink* y se muestran los resultados en la pestaña de "Configuración Hardware". Si *Brink* detecta alguna inconsistencia en el conjunto de eventos de bajo nivel, la información de la causa de dicha inconsistencia –aportada por *Brink*– se mostrará en la sección de LOG de la mencionada pestaña.

Los eventos de alto nivel son una abstracción natural de los eventos hardware. Con ellos conseguimos una representación de parámetros de rendimiento con gran poder expresivo. Un evento de alto nivel es un evento cuyo valor se calcula mediante combinación del valor de otros eventos (de alto o bajo nivel).

Cabe destacar que, mientras que los eventos de bajo nivel pueden tratarse de manera independiente, los eventos de alto nivel sólo pueden concebirse en conjunto. El hecho de que un evento de alto nivel esté compuesto por n eventos, hace que el conjunto completo de los eventos que forma un experimento de monitorización forme un grafo de dependencias.

Por motivos explicados en detalle en la sección 3.5, el grafo de dependencias debe ser acíclico. Por ello, la interfaz gráfica debe asegurar que cualquier dependencia introducida –mediante la inserción de un experimento– siga preservando la inexistencia de ciclos en el grafo.

Como hemos podido observar, las dos funcionalidades clave ofrecidas por *Brank* son servir de *frontend* para *Brink* y mantener íntegro el grafo de dependencias constituido por todos los eventos del experimento de monitorización.

3.3.2. Trabajando con *Brank*

La interfaz gráfica de *Brank*, consta de tres pestañas independientes:

- Sección de edición de experimentos y de eventos hardware
- Sección de edición de eventos de alto nivel
- Pantalla de configuración hardware

En la pestaña de edición de experimentos y de eventos hardware (figura 3.3), se permite la edición, inserción y borrado de experimentos (conjunto de eventos). También aquí se realizan las tareas de creación, eliminación y configuración de eventos hardware –dependientes de la arquitectura–.

La compleja tarea de edición de eventos de alto nivel se realiza en una pestaña aparte –configuración de eventos de alto nivel (figura 3.4)– . En esta sección también pueden configurarse las características concretas los experimentos de monitorización del kernel como umbrales de violación de QoS, acciones específicas, parámetros de gestión de la precisión, ...

El resultado de la interacción con *Brink* es la configuración hardware de los eventos de bajo nivel. Para visualizar esta configuración podemos hacerlo en la pestaña de configuración HW (dibujo 3.5) . Cualquier mensaje de error de *Brink*, es mostrado en la *consola* de la parte inferior de esta pestaña.

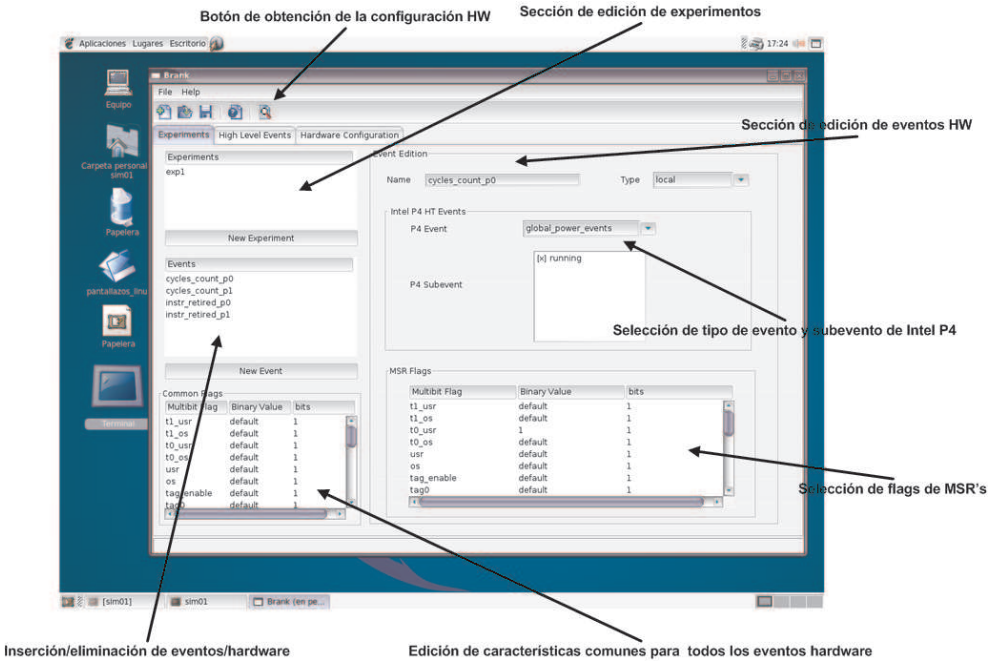


Figura 3.3: Pestaña de edición de experimentos y de eventos hardware

3.4. Diseño de Brank

En este apartado, describiremos los distintos módulos en los que se compone *Brank* y que constituyen su arquitectura. Además veremos un esquema que muestra una visión global del papel de *Brank* dentro de la infraestructura completa del proyecto.

3.4.1. Arquitectura de *Brank*

Brank es una aplicación diseñada con tecnología orientada a objetos e implementada en `C++`. En su arquitectura podemos destacar cinco subsistemas o módulos, que cubren toda la funcionalidad de la aplicación. En términos generales, los subsistemas implementan su funcionalidad a partir de un conjunto de componentes software (o clases centralizadas) que están destinados a la realización de una tarea concreta. Sin embargo, otros subsistemas implementan su funcionalidad a partir de clases que también intervienen en otros subsistemas, siguiendo un enfoque más distribuido.

Los principales subsistemas de *Brank* son los siguientes:

- Subsistema de Gestión de la Interfaz Gráfica de Usuario
- Subsistema de gestión de Experimentos de Monitorización

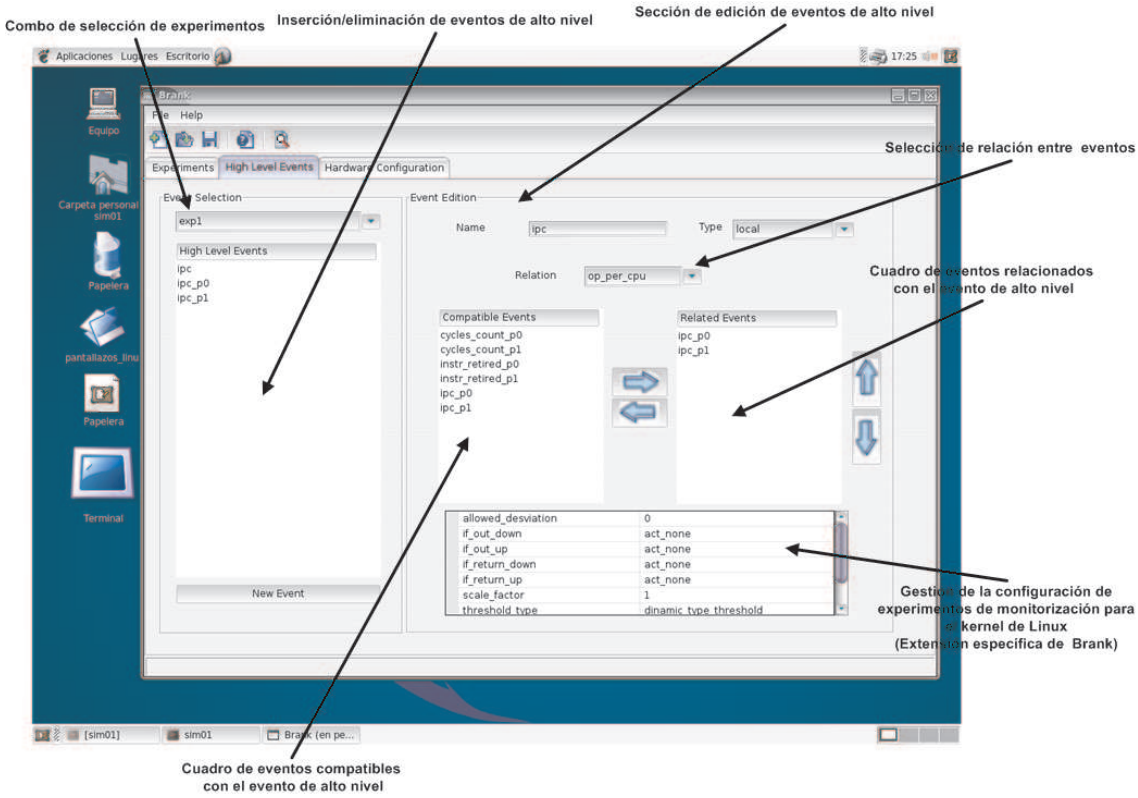


Figura 3.4: Pestaña de edición de experimentos de alto nivel

- Subsistema de Información Hardware Dependiente de Arquitectura
- Subsistema de integración Brink/Brank
- Subsistema de Generación de Configuraciones de Experimentos para el Kernel de Linux

En los posteriores apartados, describiremos con detalle la labor de cada subsistema, y cómo cada uno se relaciona con los restantes. Además se enumerarán los principales componentes que constituyen cada subsistema.

Gestión de la Interfaz Gráfica de Usuario

La interfaz de usuario de *Brank* está dividida en tres *pestañas*: dos para la configuración de eventos de alto y bajo nivel, respectivamente; y una que muestra la configuración hardware de los eventos de bajo nivel generada por *Brink*. Cada *pestaña* independiente es gestionada por un componente específico.

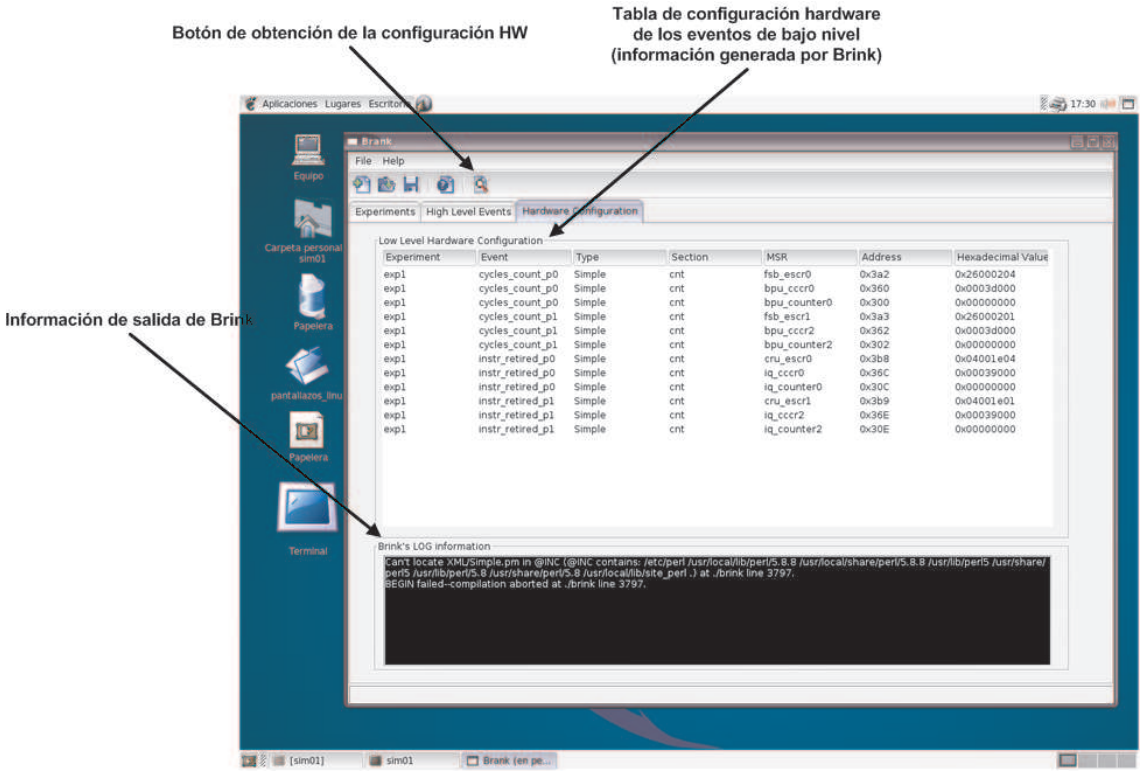


Figura 3.5: Pestaña de configuración hardware

Los gestores de cada *pestaña*, controlan la entrada del usuario y de esta manera, mantienen las restricciones necesarias para preservar la integridad de las estructuras de datos internas. Estas estructuras de datos son gestionadas por el subsistema de gestión de experimentos de monitorización.

Cabe destacar que el gestor de la pestaña de eventos de bajo nivel, está relacionado estrechamente con el susbsistema de Información Hardware Dependiente de Arquitectura. Como los eventos hardware son dependientes de arquitectura, *Brank* tiene que ofrecer soporte para gestión de eventos en diferentes plataformas hardware. Por este motivo, la interfaz gráfica debe mostrar distinta información, para configurar eventos hardware de distinta arquitectura. Esta información es proporcionada por el módulo de Información Hardware Dependiente de Arquitectura.

Este subsistema es el encargado, de realizar las tareas de E/S con ficheros, como abrir o guardar proyectos de *Brank*, importar/exportar ficheros de *Brink*, ... También es labor de este subsistema, poner en contacto al usuario con el módulo de integración con *Brink*. De esta manera, con el mero hecho de pulsar un botón, el usuario podrá actualizar instantaneamente la configuración de hardware de los experimentos actuales, que podrá ser visualizada en la *pestaña* de Configuración De Hardware.

Figura 3.6: División de *Brank* en subsistemas

Gestión de Experimentos de Monitorización

El módulo de Gestión de Experimentos de Monitorización, está constituido por un conjunto de clases, que implementan estructuras de datos y algoritmos específicos, para dar soporte a las operaciones sobre conjuntos de experimentos. Este subsistema puede verse como el *nócleo duro* de *Brank*. Sus estructuras de datos son claves para la aplicación y se describen en la sección 3.5.

Consulta de Información Hardware Dependiente de Arquitectura

Brank está concebido y diseñado, para poder ser adaptable y llegar a constituir una solución integral para la gestión de experimentos de monitorización del rendimiento para distintas arquitecturas. Por esta razón, es necesario desacoplar, en la medida de lo posible, las características y descripciones de eventos hardware propios de una arquitectura concreta.

Siguiendo está aproximación al problema, la información que describe los eventos hardware de una arquitectura se almacena en un fichero de configuración concreto –en formato XML–. El subsistema de Información Hardware Dependiente de Arquitectura debe ser un interfaz que *sirva* la información, específica de cada sistema de monitorización concreto, al resto del sistema.

Gracias a este subsistema y a una implementación específica para la lectura de la configuración hardware del fichero de descripción de cada arquitectura; puede configurarse de manera automática la sección de la interfaz de usuario para la configuración de eventos hardware (o de bajo nivel).

En la actualidad, *Brank* sólo dispone de una implementación para el sistema de monitorización de Intel Pentium IV; no obstante, el diseño permite, sin gran esfuerzo, la integración con otras implementaciones de distintas arquitecturas a modo de *framework*.

Subsistema de integración *Brink/Brank*

Este módulo de *Brank* está formado por un generador de ficheros XML para *Brink* y un procesador específico para extraer información del fichero de salida de *Brink* (fichero de LOG).

El generador de XML, no está implementado de manera centralizada sino que, la clase que representa a los eventos de bajo nivel posee métodos específicos para la generación de fragmentos del documento XML. Esta estrategia de generación XML de manera distribuida, supone una solución flexible y eficiente. La misma técnica es utilizada en la generación de ficheros de proyecto de *Brank* y ficheros de configuración de experimentos para el *kernel* de Linux.

La información generada por *Brink*, se almacena en un fichero de texto. Este fichero contiene valiosa información de configuración hardware e información específica de *Brink* que es ignorada en el proceso de análisis. La información de configuración consiste en conjuntos de ternas de nombres de registros, direcciones en hexadecimal y secuencias binarias.

El análisis y extracción de la información del fichero de texto es una tarea compleja, dada la naturaleza de la gramática subyacente a la estructura de la información del fichero. Para desarrollar el *parser*, se han empleado técnicas sistemáticas de desarrollo de procesadores de lenguaje. El algoritmo de análisis es una variante del algoritmo de *análisis descendente predictivo recursivo*, en la que se emplea un buffer acotado, para la realización del proceso de *backtracking*[3] de manera eficiente.

Las tareas de gestionar la ejecución de *Brink* con la información generada por la aplicación, así como la captura de la información que constituye la salida de *Brink*; la realiza un *script* dependiente de plataforma. La llamada al *script* específico queda determinada en tiempo de compilación.

[3]El proceso *backtracking* o vuelta atrás, es necesario para implementar el análisis del fichero ya que la gramática subyacente es $LL(k)$ con $k > 1$. Esta naturaleza de la gramática, obliga a tener una estructura temporal que almacene los *tokens* (unidades léxicas) de la entrada, que ofrezca las operaciones típicas de un iterador bidireccional y dinámico sobre el conjunto de tokens. El conjunto de tokens es producido y almacenado en el buffer a partir de la entrada.

Generación de de ficheros de experimentos para el *Kernel*

Como se comentó en la sección 3.2.2, *Brank* ha sido una herramienta concebida para servir de apoyo, al desarrollo del planificador de Linux con soporte para QoS en procesadores con SMT (objetivo final del proyecto). Para ello, la aplicación posee unas extensiones especiales para la gestión de archivos de configuración del Kernel de Linux para QoS.

Esta extensión de *Brank*, afecta a diferentes subsistemas. Para gestionar la información de manera visual, la interfaz gráfica dispone de secciones específicas para configurar parámetros como umbrales, especificación de intervalos estáticos o dinámicos,...

El módulo realiza tareas de generación XML, siguiendo la misma técnica de implementación *distribuida* que la empleada en otras ocasiones (sección 3.4.1).

3.4.2. Visión Global del Sistema

La figura 3.7, muestra una visión global de la interacción entre todos los componentes. El objetivo de la ilustración es aclarar las tareas que desarrolla cada componente y justificar las situaciones que han demandado el desarrollo de *Brank* dentro del proyecto.

Como podemos observar en la figura, el usuario interacciona con *Brank* a través de su interfaz gráfica. Mediante el uso de *Brank* pueden confeccionarse de manera sencilla, conjuntos de experimentos de monitorización que pueden guardarse y cargarse de un fichero. Como queda claro en el dibujo, la interacción con *Brink* se realiza de manera transparente al usuario a través de la aplicación.

Ya que *Brank* ha de ser una aplicación multiplataforma, hemos evitado el hecho de que *Brank*, procediera a la inserción de los experimentos directamente en el kernel –lo cual demandaría el uso de código dependiente del sistema–.

La inserción de experimentos de monitorización en el kernel, es realizada por *parser_experiments*. Se trata de una aplicación de línea de comandos, que realiza la tarea de conversión de la información de configuración del kernel –exportada por *Brank* en XML–, a una representación en C aceptada por el kernel. Una vez realizada la conversión y construcción de la estructura de datos específica, se procede a la invocación de la llamada al sistema `add_experiments()`, que realiza la inserción efectiva de los experimentos en el nócleo.

Por otro lado, cabe destacar que los ficheros de configuración del kernel generados por *Brank* constituyen una representación en XML muy próxima a la representación de los experimentos de monitorización del kernel.

Como se aprecia en la figura 3.7, el usuario puede interactuar con la interfaz ofrecida por el planificador para SMT de dos maneras. La primera, se realiza de indirectamente insertando experimentos en el nócleo a través de `parser_experiments`. La interfaz del sistema de ficheros `/proc/hypertreading` también puede utilizarse para ajustar parámetros de configuración del planificador.

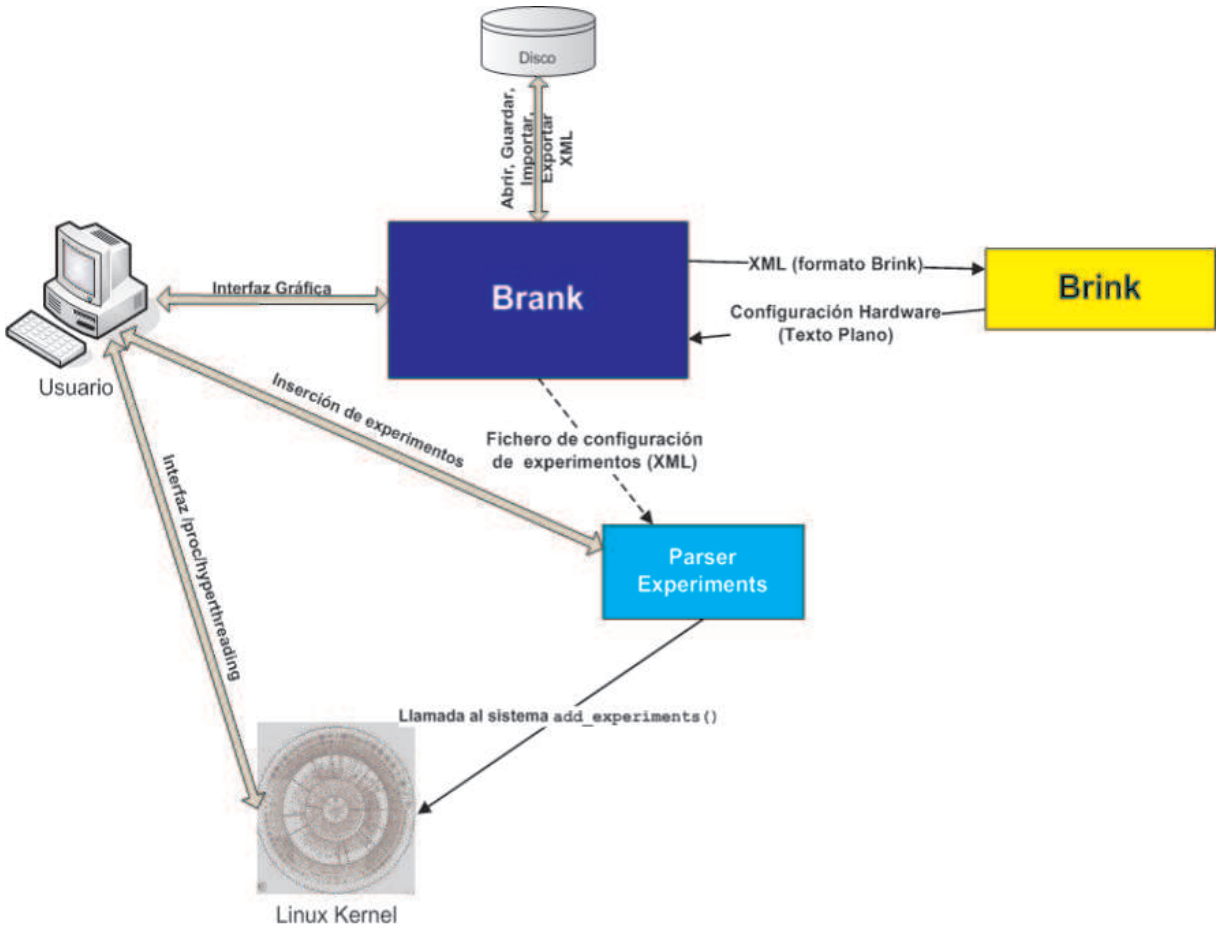


Figura 3.7: Esquema de la arquitectura global del sistema

3.5. Estructuras de datos de *Brank*

Los experimentos de monitorización, son la entidad de más peso para *Brank*. Están constituidos por conjuntos de eventos de alto y bajo nivel que están estrechamente

relacionados entre sí. Como adelantamos en secciones anteriores (sección 3.3), estas relaciones se producen por la naturaleza de representación de los eventos de alto nivel.

El conjunto de eventos hardware (o de bajo nivel), es concebido desde el punto de vista de *Brank*, como una colección de eventos independientes. Sin embargo, estos conjuntos pueden presentar inconsistencias –como indicamos en la página 45 , sección 3.1.1 – . Estas inconsistencias solo se detectan cuando se calcula la configuración hardware obtenida al interactuar con *Brink*.

Los eventos de alto nivel, son un mecanismo de mayor nivel de abstracción que los eventos hardware (o de bajo nivel). Pretenden la creación de eventos que permitan la representación de parámetros de rendimiento clásicos como el IPC, la tasa de fallos de cache, el porcentaje de aciertos de predicción de saltos, ... todos ellos incapaces de representarse a través de un ónico evento hardware. También pretenden ser un mecanismo flexible, para definir eventos como cualquier combinacion de otros n eventos y así no poner límites a la capacidad de representación.

Este convenio de representación, lleva a concebir un experimento de monitorización como un gran grafo de dependencias dirigido[4]. En el grafo, los nodos son eventos de alto y bajo nivel, y las aristas representan relaciones entre los eventos. Una arista v que va de un nodo A a un nodo B, indica que el nodo B está definido a partir del nodo A. De este modo se indica que B depende de A.

Como ningón evento de bajo nivel –representado por un nodo cualquiera L– depende de otros eventos, no existirá ninguna arista entrante a L en el grafo de dependencias. En cambio, los nodos que posean k aristas entrantes (con $k > 0$), se identificarán con eventos de alto nivel que dependen de otros k eventos (de alto o bajo nivel).

Como hemos indicado previamente, los eventos de alto nivel se definen en términos de otros eventos. Este hecho impide que se de la circunstancia de que un evento A dependa de otro evento B y, a su vez, B dependa (directa o indirectamente) de A. Esto no permite que existan ciclos en el grafo, típicos de esta situación. Por tanto, *Brank* debe preservar en la interacción con el usuario, que el grafo de dependencias entre eventos se mantenga *acíclico*.

En la figura 3.8, se describe gráficamente un experimento de monitorización con todas las relaciones entre sus eventos. Como vemos, dicho grafo es acíclico. Los nodos

[4]En un grafo dirigido, las aristas que conectan dos nodos cualquiera O y D tienen dos extremos diferenciados que indican una dirección. El extremo saliente de la arista está conectado al nodo origen O, y dicha arista es entrante a D, nodo destino.

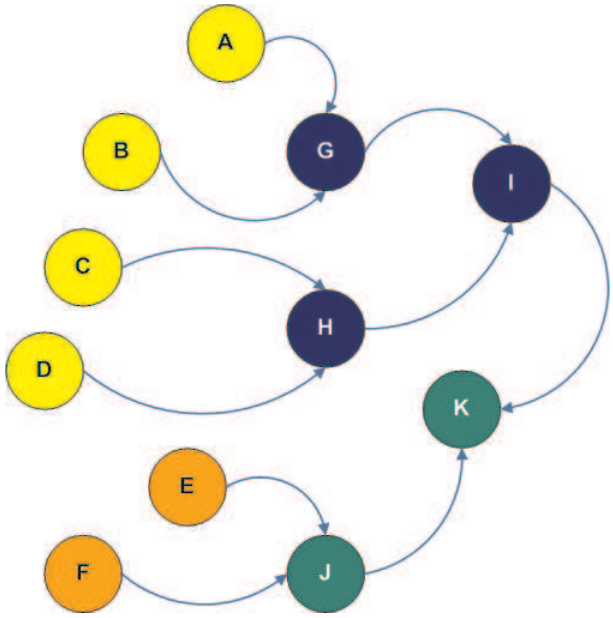


Figura 3.8: Conjunto de eventos de un experimento de monitorización

A,B,C,D,E,F se identifican con eventos hardware ya que sus nodos carecen de aristas entrantes. Las dependencias entre eventos pueden ser directas e indirectas. Se llaman directas si los dos nodos están conectados por una arista, e indirectas si existe un camino de longitud $L > 1$ entre ambos.

3.5.1. Diferencias entre la representación de los experimentos de monitorización de *Brank* y del *kernel*

La representación empleada en el kernel (descrita en la sección 3.5.3) distingue dos subtipos dentro de los eventos de alto y bajo nivel: local y global. Un evento es local, cuando su cuenta se realiza para un proceso determinado. Un evento es global cuando constituye una medida de rendimiento global del sistema –de este modo, no asociado a ningón proceso en particular–. De este modo, para el kernel, un experimento de monitorización consta de cuatro conjuntos de eventos: `ll_local`, `ll_global`, `hl_local` y `hl_global`[5].

Las operaciones que ofrece el kernel para trabajar con los experimentos en el planificador SMT son bien distintas a las soportadas desde modo usuario en *Brank*. El planificador necesita contar eventos hardware y calcular valores de eventos de alto nivel para tomar decisiones de planificación. Por otro lado, la interfaz del nócleo que ofrece

[5] El prefijo "hl" significa *High Level* –alto nivel– y "ll" representa *Low Level* –bajo nivel– .

el planificador SMT para los experimentos, sólo permite introducir y consultar experimentos completos de monitorización, así como consultar sus estadísticas asociadas.

La utilización de una estructura de grafo, para representar experimentos en el kernel, no es apropiada por motivos de eficiencia y optimización. Partiendo de la base, de que los conjuntos de eventos que constituyen el experimento, permanecen intactos durante su estancia en el kernel; debe emplearse una estructura estática que permita optimizar los algoritmos de cuenta de eventos implementados en el nócleo.

Para conseguir esta optimización y obtener algoritmos de complejidad lineal con respecto al nómero de experimentos, $O(n)$, se emplean vectores o arrays ordenados de eventos hardware. Los arrays ordenados se construyen a partir de los grafos de experimentos que utiliza *Brank* para su representación. Para efectuar la construcción de los arrays ordenados se utiliza un algoritmo específico de *ordenación topológica* para grafos *acíclicos* (descrito en la sección 3.5.3.

Los ficheros de configuración de experimentos de monitorización para el kernel, que son generados por *Brank*, se obtienen mediante la aplicación de este algoritmo de grafos. La generación manual de estos ficheros – que describen los arrays ordenados que utiliza el kernel– es una tarea compleja. Gracias a la automatización que ofrece *Brank*, la interacción con el nócleo para la inserción de experimentos resulta una tarea sencilla para el usuario.

3.5.2. Implementación de un grafo genérico en `C++`

El motivo de esta sección es describir la implementación de un grafo genérico en C++, para aclarar los algoritmos de la posterior sección –basados en esta implementación–. La implementación posee un alto grado de reutilización de componentes software ya que se han utilizado tipos de datos y contenedores ya implementados en la STL (*Standard Template Library*) de `C++`, como `string`, `map` , `list`, `set`, `vector`, `queue`...

La implementación consta de dos clases `GraphNode` y `Graph`. `GraphNode` representa a los nodos individuales del grafo y `Graph` almacena el conjunto de objetos de tipo `GraphNode` que constituyen el grafo.

Cada nodo es designado con un identificador ónico –de tipo `std::string` –. En él puede almacenarse un puntero a cualquier tipo de datos ya que `GraphNode` se implementa por medio de una `template` (plantilla). La representación de la matriz de adyacencia del grafo se lleva a cabo con listas de adyacencia en cada nodo. Para conseguir acceso

con coste $O(log(n))$ se ha utilizado un array asociativo como `std::map`, cuya implementación subyacente está basada en el uso árboles equilibrados.

En el listado 3.1 sólo se muestra la interfaz del grafo, para dar una idea general de las operaciones. En la posterior sección, se incluirá la implementación de algunos de los métodos del grafo, código clave para comprender los algoritmos específicos.

Listing 3.1: Interfaz de operaciones del grafo

```
#ifndef GRAPH_H
#define GRAPH_H
#include <map>
#include <list>
#include <string>
#include <set>
#include <queue>
using namespace std;

/*
 * Esta clase implementa el nodo de un grafo
 * como una lista de adyacencia
 */
template <class T> class GraphNode
{

public:
    /*Sección de declaración interna de tipos*/
    typedef  T dataType;
    typedef  map<string,int> mapType;
    typedef  typename mapType::iterator iteratorType;
    typedef  pair <const string,int> pairType;
private:
    //lista de adyacencia asociada al nodo
    mapType *adjacencyList;
    T *data;
public:
    //constructor por defecto
    GraphNode(T* data=NULL);
    //constructor de copia
    GraphNode(const GraphNode<T>& g);
    //Operador de asignación
    GraphNode<T>& operator=(const GraphNode<T>& g);
    //destructor
    ~GraphNode();

    //Insercion de una arista en la lista de adyacencia
    void insertEdge(const string& nodeid,int edge=1);
    //Eliminacion de una arista en la lista de adyacencia
    void deleteEdge(const string& nodeid);
    //Devuelve el coste asociado a la arista si existe
    pair<bool,int> getEdge(const string& nodeid);
    //Devuelve la lista de nodos adyacentes a otro
    void getEdgeIDs(list<string>* lista);
    //Devuelve cierto si existe 1 arista entre el nodo y nodeid
    bool containsEdge(const string& nodeid);
    //Devuelve el numero de aristas adyacentes
    int getEdgeCount() const;
    //devuelve el objeto almacenado en el nodo
    T* getData();
    //Renombrar un nodo del grafo => se renombran todas las listas de adyacencia
    void renameNode(const string& oldName,const string& newName);
};
```

```
/*
 * Esta clase reliza la implementación de un grafo dirigido basado
 * en listas de adyacencia.
 * Los nodos del grafo contienen las listas de adyacencia
 * , por lo que la matriz de adyacencia está distribuida
 */
template <class T> class Graph
{
public:
    /*Sección de declaración interna de tipos*/
    typedef  T dataType;
    typedef  GraphNode<dataType> nodeType;
    typedef  map<string,nodeType*> mapType;
    typedef  typename mapType::iterator iteratorType;
    typedef  pair <const string,nodeType*> pairType;
private:
    /*Atributos privados*/
    //Lista asociativa de nodos del grafo
    mapType *nodes;

public:
    //constructor
    Graph();
    //destructor
    ~Graph();
    //Insercion de un nodo en el grafo (id,Datos)
    void insertNode(const string& nodeid,T* data);
    //Devuelve la informacion asociada a un nodo
    T* getNode(const string& nodeid);
    //Borra el nodo con nombre nodeid
    void deleteNode(const string& nodeid);
    /*Insercion de una arista con coste = 'edge' entre origin
         y destination*/
    void insertEdge(const string& origin,int edge,const string& destination);
    /*Borrado de la arista entre origin y destination*/
    void deleteEdge(const string& origin,const string& destination);
    //Coste y presencia de la arista entre origin & destination en el grafo
    pair<bool,int> getEdge(const string& origin,const string& destination);
    //Lista de nombres de los nodos del grafo
    void getNodeIDs(list<string>* lista);
    //Lista de nombres de los nodos del grafo
    void getNodeNamesSet(set<string>* conjunto);
    //Lista de aristas adyacentes a un nodo
    void getEdgeIDs(const string& nodeid,list< string >* lista);
    //Esta o no el nodo 'nodeid'
    bool containsNode(const string& nodeid);
    //Numero de nodos de G
    int getNodeCount() const;
    //Numero de aristas de G
    int getEdgeCount() const;
    //Vaciar el grafo
    void clear();
    //Renombrar un nodo del grafo => se renombran todas las listas de adyacencia
    void renameNode(const string& oldName,const string& newName);
    //Obtener los nodos alcanzables desde un nodo
    void getReachableNodes(const string& origin, set<string> *reachableSet);
};
```

3.5.3. Algoritmos

Como comentamos en secciones anteriores, los algoritmos de grafos involucrados en la implementación de *Brank* se emplean en tareas clave que dan soporte a funcionalidades

esenciales de la aplicación.

Manteniendo un grafo *acíclico*

Una tarea primaria del sistema, es mantener íntegro el grafo que representa internamente los experimentos de monitorización. Por ello *Brank* no debe permitir la inserción de aristas –relaciones entre eventos–, que introduzcan ciclos en el grafo de dependencias.

Problema 1. *Dado un nodo O y un grafo acíclico $G \equiv< N, A >$ con N conjunto de nodos de G, A conjunto de aristas de G y $O \in N$; se busca un conjunto de nodos $U \subseteq N$ tal que $U = \{D \in N \mid G' \equiv< N, A \bigcup \{(D, O)\} >$ y G' no sea un grafo acíclico $\}$.*

Intuitivamente, la descripción del problema busca un algoritmo que toma como entrada un grafo acíclico y un nodo fijo O, y devuelva un conjunto de nodos del grafo. Suponiendo O, nodo destino fijo de una arista nueva para insertar al grafo; cada uno de los nodos D del conjunto de nodos resultado, cumplen la propiedad de que al añadir una arista de D a O, el grafo siga siendo acíclico.

La estrategia seguida para implementar este algoritmo se basa en el concepto de nodos alcanzables desde un nodo.

Definición 1. *Dado un nodo P y un grafo dirigido $G \equiv< N, A >$ con N conjunto de nodos de G, A conjunto de aristas de G y $P \in N$; se define* ***el conjunto de nodos alcanzables*** *de G a partir de un nodo O como el subconjunto $REACH(G, P) \subseteq N$ tal que $REACH(G, P) = \{Q \in N \mid$ existe un camino que va desde P hasta Q $\}$*

La solución que buscamos al problema, puede resolverse de manera sencilla con este nuevo concepto. Es fácil probar que el conjunto solución S que devuelve el algoritmo especificado en el problema 1 es $S \equiv N - REACH(G, P)$.

Ahora el problema se reduce al cálculo de los nodos alcanzables desde un nodo de un grafo dirigido. Una vez obtenido, dicho conjunto se resta de manera conjuntista al total de nodos para así obtener la solución.

El algoritmo de calculo de los nodos alcanzables a partir de un nodo del grafo, está implementado como un método de la clase `Graph`. El código puede verse en el listado 3.2.

Listing 3.2: Algoritmo de cálculo de los nodos alcanzables

```
//Obtener los nodos alcanzables desde un nodo
```

```
template <class T>
void Graph<T>::getReachableNodes(const string& origin, set<string> *reachableSet)
{
    //Inicializacion de marcadores
    map<string,bool> visitado;
    //Cola de nodos para hacer el recorrido
    queue<string> cola;
    //recorremos nodos
    iteratorType it;
    for (it= nodes->begin(); it!=nodes->end();++it)
    {
        visitado[it->first]=false;
    }

    //suponemos visitado el primero
    visitado[origin]=true;
    //insertamos el primero
    cola.push(origin);
    //iteramos mientras la cola no este vacia
    while(!cola.empty())
    {
        //cogemos el primero y trabajamos con el grafo inverso
        string next=cola.front();
        //eliminamos el primero
        cola.pop();
        //procesamos los adyacentes a este
        list<string> adyacentes;
        getEdgeIDs(next,&adyacentes);
        //ahora, recorremos la lista
        for (list<string>::iterator it=adyacentes.begin();it!=adyacentes.end();++it)
        {//solo expandimos los que no han sido visitados
            string cur=*it;
            if (!visitado[cur])
            { //expansion de nodo
                visitado[cur]=true;
                cola.push(cur);
                //añadimos al conjunto de los adyacentes al hacer la expansion
                reachableSet->insert(cur);
            }//fin expansion de nodo
        }//fin recorrido de adyacentes
    }//fin while
}
#endif // GRAFO_H
```

Para proceder al cálculo de dicho conjunto, realizamos un recorrido en anchura del grafo mediante el empleo de una cola (`std::queue`), con la que preservamos el orden de procesamiento de los nodos del grafo.

El código encargado de efectuar la resta conjuntista, está incluido como un método de la clase `HighLevelEvent`; clase que implementa toda la funcionalidad de los eventos de alto nivel. En este fragmento se buscan los eventos compatibles para poder definir un evento de alto nivel a partir de otros. El código es el siguiente:

Listing 3.3: Algoritmo de cálculo de los eventos compatibles

```
//Metodo de obtencion de los eventos compatibles
void  HighLevelEvent::getCompatibleEvents(set<string> *conjunto)
{
    string nodename=TO_STD_STRING(getName());
```

```
    set<string> reachableNodes;
//Algoritmo del grafo
    grafoInverso->getNodeNamesSet(conjunto);
    grafoInverso->getReachableNodes(nodename,&reachableNodes);
//Ahora hacemos la resta de estos dos conjuntos
    for (set<string>::iterator it=reachableNodes.begin();it!=reachableNodes.end();++it)
    {
        //quitamos ese elemento
        conjunto->erase(*it);
    }
//Eliminamos tambien a él mismo
    conjunto->erase(nodename);
}
```

Algoritmo de *ordenación topológica* de un grafo *acíclico*

La conversión entre la representación empleada por *Brank* y las estructuras de datos utilizadas por el kernel, para el almacenamiento de los experimentos de monitorización es una tarea clave de *Brank*. La conversión de ambas representaciones consiste en la realización de una partición en 4 vectores de eventos (representación del *kernel* para los experimentos), del grafo de dependencias construido por *Brank*.

Los cuatro vectores han de estar ordenados[6] siguiendo un criterio de dependencia. El criterio de dependencia que controla la ordenación es muy sencillo: si un experimento A depende directa o indirectamente de otro evento B, B deberá estará situado antes que A en el array.

El proceso de construcción de los vectores es sistemático. En primer lugar se realiza una partición del grafo global en cuatro subgrafos, con nodos disjuntos, que constituirán los elementos de cada vector. Estos subgrafos se tratan como vectores desordenados que tenderemos que ordenar. La ordenación de cada vector la determina el vector global que se obtiene como resultado de la aplicación del algoritmo de *ordenación topológica* al grafo total.

Puede emplearse cualquier algoritmo de ordenación conocido para realizar esta tarea. El aspecto importante es cómo definamos el operador $<$ que controla el proceso de ordenación. Afortunadamente, podemos dar una definición de $<$ (*menor que*) para los nodos de un grafo acíclico que verifica la siguiente propiedad:

Propiedad 1. *Dado un dominio D, Se dice que la operación relacional $<$ (menor que) sobre D está bien definida $\Longleftrightarrow_{def}$ se cumple que $\forall a, b \in D$ si $a < b$ entonces $b \not< a$*

[6] El algoritmo de cuenta de eventos del kernel, parte del supuesto de que los vectores se encuentran ordenados.

Definición 2. *Sea un grafo acíclico $G \equiv < N, A >$ con N conjunto de nodos de G, A conjunto de aristas de G. Fijado el dominio N de nodos de un grafo acíclico, sean n_1 y n_2, dos nodos de G definimos la operación relacional $<$ sobre N como:*

$$n_1 < n_2 \iff_{def} n_2 \in REACH(G, n_1)$$

Colorario 1. *La operación '$<$' para grafos acíclicos, descrita en la definición 2, está bien definida. Sea $G \equiv < N, A >$ un grafo acíclico. Procedemos por reducción al absurdo:*

Demostración

Sunpongamos que $\exists\ a, b \in N$ *t.q.* $a < b$ *y* $b < a$
$\iff b \in REACH(G, a)$ *y* $a \in REACH(G, b)$
$\iff$ *Existe un camino que va de a hasta b y*
existe otro camino que va de b hasta a
$\iff_{def}$ *Existe un ciclo en el grafo*
$\Rightarrow$ $\sharp$ *Llegamos a una contradicción pues el grafo es acíclico*

Intuitivamente, la definición anterior describe la operación $<$ para los nodos de un grafo acíclico. Así pues se cumple que todo P del conjunto de nodos alcanzables (definición 1) a partir de otro Q, satisfacen que $Q < P$.

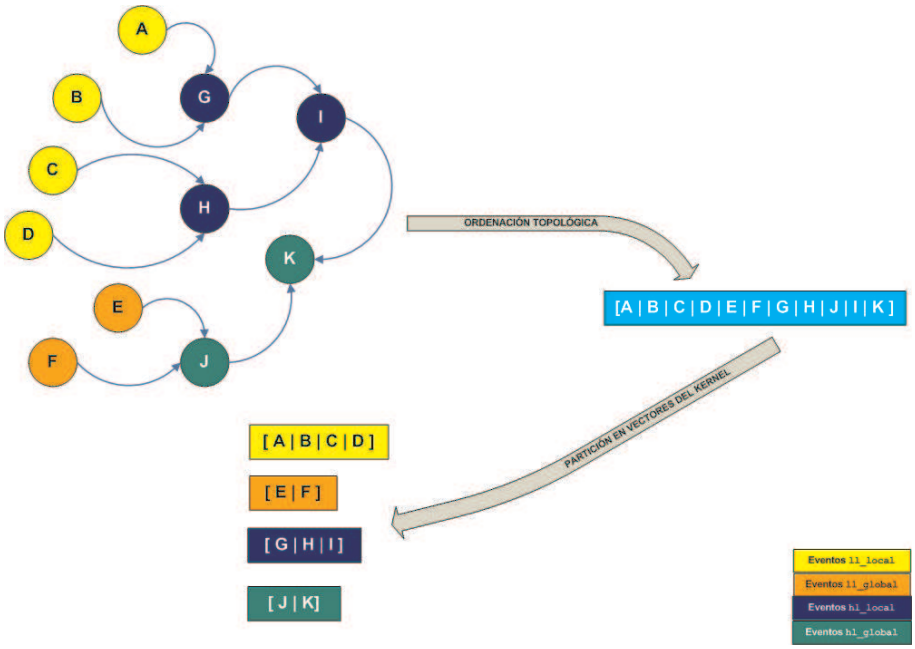


Figura 3.9: Aplicación del algoritmo de ordenación topológica a un experimento de monitorización

Una vez definida la operación *menor que*, ya podemos proceder a implementar el algoritmo de *ordenación topológica*. La figura 3.9 muestra un ejemplo de la ejecución del algoritmo de ordenación topológica aplicado a todo el grafo. Posteriormente se realiza la partición en los *arrays* demandados por el *kernel*.

En la implementación se utiliza `sort()`, un algoritmo genérico basado en *quicksort* que proporciona la librería estándar de plantillas de `C++`, STL. `sort()` recibe como parámetros el vector a ordenar (`std::vector`) y un objeto que defina el operador <.

Listing 3.4: Clase que implementa la operación < ” requerida por `sort()`

```
  /* ** Clase auxiliar para definir el operador '<'
   * necesario para efectuar la ordenacion topologica
   * del grafo
   */
  class HighLevelEvent::Cmp
    {
    private:
        //Contiene punteros a los cierres de cada nodo
        //=> nodos alcanzables;
        //grafo
        Graph<Event> *graph;
                //Tabla de cierres precalculada
        map<string,set<string>*> *reachableNodes;
    public:
        //Contructor => el grafo o conjunto de eventos =>
        Cmp(Graph<Event> *graph);
                /* ** metodo explícito para liberar memoria **
                 * requerido por sort() de STL
                 */
                void free();
        /* ** operador efectivo de comparacion **
         * Se pasan los identificadores de los nodos
         */
                bool operator()(const string& node1,const string& node2);
    };

//Contructor => el grafo o conjunto de eventos =>
HighLevelEvent::Cmp::Cmp(Graph<Event> *_graph)
        :graph(_graph)
{
    //Calculo de los cierres de todos los nodos
    reachableNodes=new map<string,set<string>*>();
    list<string> lista;
    graph->getNodeIDs(&lista);
    for (list<string>::iterator it=lista.begin();it!=lista.end();++it)
    {
        string nodename=*it;
        set<string> *nodosAlcanzables=new set<string>();
        graph->getReachableNodes(nodename,nodosAlcanzables);
        //insertamos en la tabla de conjuntos
        (*reachableNodes)[nodename]=nodosAlcanzables;

    }//finfor
}


/* ** metodo explícito para liberar memoria **
* requerido por sort() de STL
*/
void HighLevelEvent::Cmp::free()
{
```

```
    //liberacion de los conjuntos de cierre
    if (reachableNodes!=NULL)
    {
        for (map<string,set<string>*>::iterator it=reachableNodes->begin();
                it!=reachableNodes->end();
                ++it)
        {
            set<string> *nodosAlcanzables=it->second;
            delete nodosAlcanzables;
        }//finfor
        //borramos la estructura global
        delete reachableNodes;
        reachableNodes=NULL;
    }

}


 /* ** operador efectivo de comparacion **
         * Se pasan los identificadores de los nodos
         */
bool HighLevelEvent::Cmp::operator()(const string& node1,const string& node2)
{
    //Metodo < de comparacion de 2 eventos
    //un evento es menor que otro si
    // el el segundo evento esta en el cierre del primero
    map<string,set<string>*>::iterator it=reachableNodes->find(node1);
    //CASO: si existe el cierre de nodo1
    if (it!=reachableNodes->end())
    {
        set<string>* cierre_nodo_1= it->second;
        //
        set<string>::iterator it2=cierre_nodo_1->find(node2);
        //CASO: nodo2 esta en el cierre de nodo1 (nodo1 depende de nodo2)
        if (it2!=cierre_nodo_1->end())
            return true;
        else
            //CASO: nodo2 no esta en el cierre de nodo1 (nodo1 no depende de nodo2)
            return false;
    }
    //CASO: no existe el cierre de nodo1
    else
        return false;
}
```

Listing 3.5: Algoritmo de ordenación topológica

```
/* *** Algoritmo de ordenación de un experimento ***
 * Su trabajo consiste en asignar a eventos de alto y bajo nivel
 * un número entero que ocupa su posición en el array respectivo
 * que ocupará el evento dentro del núcleo
 */
void Experiment::sortEvents()
{
        //Declaración de los 4 vectores temporales
    vector<string> ll_local,ll_global,hl_local,hl_global;
    list<string> strList;
    //Obtengo los nombres de los nodos
    eventos->getNodeIDs(&strList);

        /*Este bucle realiza las tareas de volcado del nombre del
                evento a cada array que corresponda*/
    for (list<string>::iterator it=strList.begin();it!=strList.end();++it)
    {
        string name_evt=*it;
        Event* event=eventos->getNode(name_evt);
```

```
        if (event->getLevelMode()==Event::LL)
        {
            if (event->getThreadMode()==Event::TS)
                ll_local.push_back(name_evt);
            else
                ll_global.push_back(name_evt);
        }
        else
        {
            if (event->getThreadMode()==Event::TS)
                hl_local.push_back(name_evt);
            else
                hl_global.push_back(name_evt);
        }
    }//fin for

//Instanciación del comparador global (operador < para la ordenación)
    HLComparator hlc(grafoInverso);

//Ordenacion de los 2 vectores usando el mismo comparador => los ll estan ordenados
    sort(hl_local.begin(),hl_local.end(),hlc);
    sort(hl_global.begin(),hl_global.end(),hlc);

/* **Proceso de Etiquetado**
 * Una vez procedida la tarea de ordenación
 * se asigna a cada evento el lugar que ocupa en su array respectivo
 */
    int cnt=0;

    for (vector<string>::iterator it=hl_local.begin();it!=hl_local.end();++it,cnt++)
    {
        grafoInverso->getNode(*it)->setIndex(cnt);
    }
    cnt=0;
    for (vector<string>::iterator it=ll_local.begin();it!=ll_local.end();++it,cnt++)
    {
        grafoInverso->getNode(*it)->setIndex(cnt);
    }
    cnt=0;
    for (vector<string>::iterator it=hl_global.begin();it!=hl_global.end();++it,cnt++)
    {
        grafoInverso->getNode(*it)->setIndex(cnt);
    }
    cnt=0;
    for (vector<string>::iterator it=ll_global.begin();it!=ll_global.end();++it,cnt++)
    {
        grafoInverso->getNode(*it)->setIndex(cnt);
    }

    //liberar hlc
    hlc.free();
}
```

Capítulo 4

Implementación

Como base para la implementación del prototipo propuesto se han tomado las mediciones obtenidas mediante los contadores hardware anteriórmente descritos. Usando como mecanismo de monotorización los contadores de la arquitectura $IA-32$ se han realizado mediciones de eventos relevantes para estudiar el rendimiento del sistema.
Se ha desarrollado una amplia interfaz para permitir el fácil manejo de estos contadores. Incluso un usuario con escasos conocimientos acerca de estos podría configurarlos para medir eventos en su sistema.

4.1. Estructura de los módulos de código del prototipo

Debido a la amplitud del código fuente generado para el prototipo, se ha estructurado el mismo en diversos módulos que se enumeran a continuación. Se trata de archivos que contienen las estructuras de datos y funciones de las que posteriormente hablaremos de una manera organizada.

- ***ht_scheduler.c***, ***ht_scheduler.h***. Contienen:
  - Variables globales para el profiling del sistema y el algortimo de Calidad de Servicio.
  - Funciones que inicializan todas las estructuras de datos de los experimentos y de las estadísticas.
  - Implementación del algoritmo de la política Calidad de Servicio bajo la función *ht_keep_qos()*.
  - Implementación de las funciones *do_count_local_events()* y *do_count_global_events()* que actualizan las mediciones de los experimentos tanto locales como globales y sus estadísticas

  Son por tanto el módulo principal de la implementación del prototipo porque manejan los datos globales para las operaciones de profiling y calidad de servicio.

- ***ht_experiments.c***, ***ht_experiments.h***. Contienen:
  - Funciones que chequean las estructuras de datos introducidas con la llamada al sistema (valida si contienen experimentos construidos corréctamente y detecta inconsistencias).
  - Implementación de las funciones que gestionan un experimento en concreto: inicialización y búsqueda de operandos (para el caso de los de alto nivel).

- Implementación de las función *doCount()* que recorre los arrays de experimentos de bajo y alto nivel actualizando uno a uno los resultados correspondientes.

En resumen, contienen las funciones necesarias para gestionar individualmente los experimentos insertados.

- ***ht_interface.c***, ***ht_interface.h***. Contienen:

  - Función de inicialización *__init __hta_proc_start()* de los nodos introducidos en el */proc*.

  - Implementación de las funciones que gestionan la lectura y escritura de las entradas adicionales creadas en */proc*.

  - Implementación de la llamada al sistema *sys_add_experiments()* que permite añadir las estructuras de datos que contienen los experimentos y que han sido generadas en modo usuario.

Constituye en sí la interfaz de comunicación del prototipo con el exterior. Las funciones que implementa permiten comunicar al usuario con el sistema para visualización de resultados y depuración y al sistema con el usuario para la configuración de parámetros relevantes del mismo.

- ***ll_events.h***. Contiene todas las estructuras de datos relacionadas con los eventos de bajo nivel y con los contadores hardware.

- ***ll_events_inline.h***. Contiene la interfaz de operaciones directamente relacionadas con los eventos de bajo nivel. También implementa la interfaz de manejo de los contadores hardware. Puesto que la frecuencia a funciones de esta interfaz es muy alta, se ha propuesto hacer estas últimas operaciones como *inline*.

- ***hl_events.h***. Contiene todas las estructuras de datos relacionadas con los eventos de alto nivel.

- ***pmc_asm.h***. Contiene la implementación de las macros de gestión directa de los contadores (ensamblador en línea).

- ***pmc_const.h***. Define todos las constantes de los contadores hardware para la familia *Pentium 4 e Intel Xeon* (direcciones de todos los contadores y máscaras frecuentes para algunos de ellos).

- ***pmc_asm.h***. Contiene la implementación de las macros de gestión directa de los contadores (ensamblador en línea).

Además, nos hemos visto obligados a modificar fuentes relevantes del kernel, que se enumeran a continuación:

- ***linux-2.6.13/kernel/sched.c***: contiene todas las implementaciones relacionadas con el planificador. En concreto, se ha modificado la función *rebalance_tick()* para realizar las mediciones en cada tick de reloj (o cada cierto número de ticks, lo cual es configurable a través como se explica en el capítulo 5.8) y la función *sched_init* que inicializa el planificador.

- ***linux-2.6.13/include/linux/sched.h***: puesto que aquí se define la estructura *task_struct* hemos introducido modificaciones sobre los campos de la misma, ya que hemos tenido que añadir información perteneciente a cada proceso, como son las estadísticas de últimos valores de ipc para la calidad de servicio.

- ***linux-2.6.13/kernel/fork.h***: por las mismas razones que el fuente anterior hemos modificado la función de creación de un hijo para que se inicialicen correctamente los nuevos campos introducidos.

4.2. Estructuras de datos relevantes en el kernel

4.2.1. Estructuras de datos para los experimentos de bajo nivel

Las estructuras de datos que se han implementado para manejar dentro del kernel la información de los eventos de bajo nivel se encuentran implementadas en los archivos *ll_events.h* y las direcciones de los registros MSR y PMS están en *pmc_const.h*.

- Definiciones de los tipos para 32 bits y 64 bits

```
typedef unsigned int _uint32_t;

typedef struct{
    _uint32_t low;
    _uint32_t high;
}_uint64_t;
```

- Estructuras para albergar la información de pmc y msr

```
typedef struct {
  unsigned long address;
  _uint64_t reset_value;
  _uint64_t new_value;
} _msr_t;
```

```
typedef struct {
   unsigned long  pmc_address;
    _msr_t msr;
} _pmc_t;
```

- Estructuras para albergar la información completa de un experimento concreto. Recordemos que según la naturaleza del evento a contar, su correspondiente experimento es de tipo simple, tagging o pebs.

```
typedef struct{
  _pmc_t pmc;
  _msr_t cccr;
  _msr_t escr;
} simple_exp;
```

```
typedef struct{
    simple_exp event;
    simple_exp tag;
} tagging_exp;
```

```
typedef struct{
   simple_exp event;
   unsigned long pebs_enable;
   unsigned long pebs_matrix;
 } pebs_exp;
```

  - simple_exp: necesita únicamente un un pmc, un registro escr y un registro cccr.
  - tagging_exp: está formado por dos experimentos simples porque necesita uno de ellos para el conteo del evento y otro para el marcado.
  - pebs_exp: además de la información de un experimento simple necesita las máscaras para los registros pebs_enable y pebs_matrix, puesto que para medir en modo preciso es necesario configurar ambos.

- Estructuras para albergar la información completa de un experimento genérico.

```
typedef struct {
 char id[MAX_EXP_ID];
 union {
        simple_exp s_exp;
        tagging_exp t_exp;
        pebs_exp p_exp;
 }g_event;
 int type;
}low_level_exp;
```

  - id: nombre que se le da al experimento en el sistema.
  - g_event: información del evento del que se encarga este experimento de bajo nivel.
  - type: naturaleza simple (type toma valor 0), tagging (type toma valor 1) o pebs (type toma valor 2) del evento.

4.2.2. Estructuras de datos para los experimentos de alto nivel

Las estructuras de datos que se han implementado para manejar dentro del kernel la información de los eventos de alto nivel se encuentran implementadas en el archivo *hl_events.h*.

- Estructuras para albergar la información de los operadores y operandos de los experimentos (de bajo o alto nivel) con que se componen un experimento de alto nivel.

```
typedef enum {
 op_division,
 op_multiplication,
 op_sum,
 op_substract,
 op_rate,
 op_none,
 _AVAILABLE_RELATIONS
} relation_mode;
```

```
typedef enum {
 hl_local,
 hl_global,
 ll_local,
 ll_global
} relation_arg_mode;
```

```
typedef enum {
      local,
      global
    } hl_exp_mode;
```

```
typedef struct {
        relation_arg_mode mode;
        unsigned int index;
      } relation_arg_t;
```

  - mode: indica el tipo del operando. Los disponibles son bajo nivel local, bajo nivel global, alto nivel global y alto nivel local.
  - index: indice del operando que tiene en su array de experimentos. Con este índice y con el campo mode anterior localizar el valor de los operandos se hace de manera directa, sin tener que recorrer ninguno de los arrays. El campo mode distingue en qué array de experimentos del kernel hay que buscar y el campo index en qué posición.

- Estructuras para mantener la información de los umbrales impuestos a un experimento de alto nivel.

```
typedef enum {
  static_type_threshold,
  dinamic_type_threshold
} type_threshold;
```

```
typedef enum {
  act_none,
  act_up_ht,
  act_down_ht,
  _AVAILABLE_HT_ACTIONS
} action_type;
```

```
typedef struct{
  action_type if_return_up;
  action_type if_return_down;
  action_type if_out_up;
  action_type if_out_down;
}action_list;
```

  - action_list: permite distinguir qué acción se asocia a cada una de las cuatro situaciones posibles que pueden presentarse cuando se considera un umbral. Si el valor sobrepasa los límites deseados por arriba se aplicará la acción indicada en *if_out_up*. Si vuelve dentro de esos límites se aplica *if_return_up*. Si el valor sobrepasa los límites deseados por abajo se aplicá la acción impuesta en el campo *if_out_down*. Si vuelve dentro de ese rango se aplica *if_return_down*. Las acciones disponibles a aplicar son *act_up_ht* (activar el hyperthreading), *act_down_ht* (desactivar el hyperthreading) y *act_none* (no realizar ninguna acción).

- Estructuras para mantener la información de un evento de alto nivel.

```
typedef struct{
  relation_mode mode;
  relation_arg_t exp1;
  relation_arg_t exp2;
  unsigned long scale_factor;
```

```
typedef struct{
    char id[MAX_EXP_ID];
    hl_descriptor descriptor;
    unsigned long count;
    } high_level_exp;
```

```
  type_threshold  threshold_type;
  unsigned long allowed_desviation;
  action_list action_set;
} hl_descriptor;
```

- id: nombre que se le da en el sistem al experimento de alto nivel.
- descriptor: alberga el operador que se aplica a los subexperimentos con los que se calcula, la información sobre cada uno de los subexperimentos que forman sus operandos (como máximo dos), el tipo de factor de escala que se aplica al cálculo (valor entero que multiplica al resultado de la operación), el tipo de umbral que se aplica (estático o dinámico), desviación permitida respecto al valor del umbral y acciones asociadas a las situaciones posibles en el cáculo con umbrales.
- count: guarda el último valor calculado.

4.2.3. Estructuras de datos para los experimentos en el kernel

```
typedef struct {
 unsigned int   size;
 low_level_exp  array[MAX_LL_EXPS];
} ll_exps_set;

typedef struct {
 ll_exps_set ll_local_set;
 ll_exps_set ll_global_set;
 hl_exps_set hl_local_set;
 hl_exps_set hl_global_set;
} ht_experiment_t;
```

```
typedef struct{
 unsigned int    size;
 high_level_exp  array[MAX_HL_EXPS];
}hl_exps_set;
```

El número de experimentos de bajo nivel en el sistema no puede superar los 18 debido a limitaciones de los contadores hardware. Se ha limitado el número de experimentos de alto nivel hasta 20 globales y 20 locales. Son valores bastante altos que corresponden a 40 posibles combinaciones de experimentos para estudiar.

- ll_local_set: array de experimentos de bajo nivel globales.
- ll_global_set: array de experimentos de bajo nivel locales.
- hl_local_set: array de experimentos de alto nivel locales.
- hl_global_set: array de experimentos de alto nivel globales.

4.2.4. Estructuras de datos para las estadísticas medidas en el kernel

Las estructuras de datos para contabilizar las estadísticas están implementadas en el fichero *ht_scheduler.h*.

```
        typedef struct {                          typedef   struct {
         unsigned long last_value;                 statisticks_struct
         unsigned long last_value_not_zero;             ll_statistics[MAX_LL_EXPS];
         unsigned long long int acum;              statisticks_struct
         unsigned long max;                             hl_statistics[MAX_HL_EXPS];
         unsigned long min;                       } prof_struct;
         ht_cbuffer samples;
         unsigned long average;
         unsigned long old_average;
         unsigned long total_samples_counted;
         } statisticks_struct;

        prof_struct global_prof;
        //Valores de estadisticas para experimentos globales

} statisticks_struct;
```

- last_value: corresponde al útlimo valor medido del experimento.
- last_value_not_zero: corresponde al último valor no nulo medido del experimento.
- acum: acumulación de todas las mediciones realizadas desde que se configuró el experimento. Se han previsto los debordamientos en este campo implementándolo de un tipo adecuado.
- max, min: corresponden al máximo y al mínimo valor medido.

Además, para el caso de la calidad de servicio es necesario almacenar una breve historia delas mediciones del evento respecto al cual se configure el algoritmo de la política de Calidad de Servicio. Por ejemplo, si en el sistema se está haciendo profiling para el ipc local y la tasa de fallos de primer nivel local, cada tarea guarda sus estadísticas en el campo *prof*. Además, si se tiene configurado el sistema para Calidad de Servicio orientada al ipc, en dicho campo *prof* estará la información sobre el experimento que necesite el algoritmo.
Esta información es la que se muestra a continuación.

- samples: buffer circular con las últimas mediciones del valor respecto al cual se hace la calidad de servicio.
- average: media actual de los elementos que están en el buffer.
- old_average: media anterior a la de average.
- total_samples_counted: contador de cuantas veces se actualizaron las estadisticas del buffer.

```
typedef struct {
 unsigned long data[HT_MAX_CBUFFER_SIZE];
 unsigned int head;
 unsigned int size;
 unsigned int max_size;
}ht_cbuffer;
```

- data: buffer circular.

- head: índice de la cabeza del buffer. Su valor está entre 0 y MAX_SAMPLES-1.
- size: cantidad de muestras tomadas en el buffer.
- max_size: valor máximo de muestras que se pueden guardar a la vez en el buffer. Determina el tamaño de la ventana que constituye la historia relevante que influirá en la política de Calidad de Servicio. Es un valor configurable como se muestra en el capítulo 5.8.

Además se ha implementado una estructura que guarda arrays de alto nivel y bajo nivel juntos. Habrá una estructura de este tipo para cada tarea (para los experimentos locales) y otra única para el sistema (para los experimentos de carácter global).
La de cada tarea debe introducirse como un campo nuevo *prof* en su estructura *task_t* como se muestra en el siguiente fragmento de código del archivo *shed.h*:

```
struct task_struct {
 prof_struct prof;
 volatile long state;
 struct thread_info *thread_info;
 atomic_t usage;
 unsigned long flags;
 unsigned long ptrace;
 int lock_depth;
...
}
```

Para guardar los resultados que el usuario que ha configurado la herramienta como profiler quiere recibir es necesario crear una estructura más.

```
typedef struct {
 int oc_prof_buffer ;
 pid_t pid_arr[PROF_BUF_SIZE];
 int index_event_arr[PROF_BUF_SIZE];
 unsigned long last_value_arr[PROF_BUF_SIZE];
 struct rw_semaphore sem;
} ht_buffer_prof_t;
```

- pid_arr: array para guardar el pid del proceso monitorizado en cada momento. En el caso de que el profiling sea global (no hay pid asociado), por convenio se introducirá un -1 en su lugar.

- index_event_arr: array de índices de experimentos monitorizados. Siempre corresponde a un índice del array de experimentos de alto nivel.

- last_value_arr: array de las mediciones tomadas.

- sem: semáforo para proteger los accesos concurrentes a la estructura.

- oc_prof_buffer: nivel de ocupación del triple buffer.

Para cada posición i-ésima ($0 \leq i \leq oc\_prof\_buffer$) se verifica que last_value_arr[i] tiene el valor medido del experimento de alto nivel de índice index_event_arr[i] para el proceso de pid pid_arr[i]. Como a la función que analice este buffer se le pasará el pertinente array de experimentos de alto nivel, será directo obtener el nombre o id de dicho experimento.

En el sistema mantendremos una estructura de este tipo por cada cpu (en nuestro caso serán dos). Están declaradas en *ht_scheduler.c*

```
ht_buffer_prof_t buffer_prof[NR_CPUS];
```

4.3. Interfaz desarrollada para operar con los Contadores Hardware IA-32

El manejar los contadores hardware exige hacer complejas llamadas a ensamblador para una simple pero frecuente operacion como por ejemplo la lectura de un contador. Se ha desarrollado en el prototipo una interfaz de funciones que permiten gestionar los contadores y basándose en ellas y subiendo un nivel más, otro conjunto de funciones que permiten gestionar directamente los experimentos insertados en el kernel.

4.3.1. Funciones para operar con los contadores HW

Las funciones mencionadas se encuentran implementadas en los archivos *ll_events.h*, *ll_events_inline.h* y *pmc_asm.h*.

- **void initMSR(_msr_t* handler)**: pone a cero los campos high y low de los valores reset_value y new_value. Fija el valor de reseteo del contador.
- **void initPMC(_pmc_t* handler)**: llama a la anterior para su campo msr.
- **void resetMSR(_msr_t* handler)**: resetea el msr con el valor fijado de reseteo. Este valor se fijó en reset_value con la llamada a initMSR.
- **void writeMSR(_msr_t* handler)**: escribe en el msr el valor fijado en el campo new_value.

- **void readMSR(_msr_t* handler)**: hace lectura del msr.
- **void resetPMC(_pmc_t* handler)**: resetea el pmc (haciéndolo para su msr) con el valor establecido.
- **void readPMC(_pmc_t* pmc_handler)**: hace lectura del valor actual del pmc.
- **void fastreadPMC(_pmc_t* pmc_handler)**: hace lectura rápida (los 32 bits menos significativos) del valor actual del pmc.

4.3.2. Funciones para operar con experimentos de bajo nivel

- **void init_low_level_exp(low_level_exp* exp, const char *name,int type)**: inicialización genérica del experimento para darle nombre al mismo y clasificarlo en tipo simple, tagging o pebs .
- **void __startCount(low_level_exp* exp)**: configuración del inicio de la cuenta para el experimento.
- **void __stopCount(low_level_exp* exp)**: configuración de la parada para el contador del experimento.
- **void __clearCount(low_level_exp* exp)**: configuración de reseteo para el contador del experimento.
- **void __readCount(low_level_exp* exp)**: lectura del valor del contador del experimento.
- **unsigned int __getEventCPU(low_level_exp* exp)**: análisis de la máscara configurada en escr para obtener si el evento está configurado para medirse en la cpu cero, la cpu uno o ambas.
- **unsigned int __getLastValue(low_level_exp* exp)**: obtención del último valor medido para el experimento.
- **int check_ll(low_level_exp* exp)**: chequea la integridad de los campos de un experimento de bajo nivel. Se comprueba que los registros ESCR, CCCR, PM y MSR que se referencien sean correctos.

4.3.3. Funciones para operar con experimentos de alto nivel

- **void get_operands(high_level_exp* hle, ht_experiment_t* experiments, perand *operands ,unsigned int num_ops)**: devuelve un array con num_ops valores de operandos. La información sobre cuáles son los operandos cuyo valor debe buscar se obtiene explorando los campos del experimento de alto nivel y los

valores en los arrays de experimentos. Recordemos que esta última medición se guardó en la estructura del experimento para evitar el acceso a las estructuras de estadísticas.

- **void compute_value(high_level_exp* hle,ht_experiment_t* experiments)**: calcula el valor del experimento aplicando el operador correspondiente a partir del valor de los operandos obtenido con *get_operands*
- **int check_hl(high_level_exp* exp,ht_experiment_t* ht_experiment)**: chequea la integridad de los campos de un experimento de alto nivel, esto es que los operandos referenciados existan y que el resto de campos valores configurables (tipo de operador, tipo de umbral etc) estén entre los disponibles.
- **char* print_high_level_exp(high_level_exp* hle,ht_experiment_t* experiments,char* buffer)**: devuelve una recopilación de la información para un experimento de alto nivel.

4.3.4. Funciones para operar con el conjunto de experimentos

- **void init_ht_experiment_t(ht_experiment_t* ht_experiment)**: resetea la estructura de los cuatro arrays de experimentos con una ocupación cero. Pone a cero los campos de ocupación de los arrays de experimentos del kernel.
- **void initial_configuration_MSRS(ht_experiment_t* ht_experiment)**: inicializa los registros MSR globales PEBS_ENABLE y PEBS_MATRIX_VERT antes de comenzar la cuenta. También inicia los contadores hardware para que efectuen su primera cuenta.
- **int check_ht_experiment(ht_experiment_t* ht_experiment)**: chequea la integridad de los cuatro arrays de experimentos del sistema llamando al chequeo de cada experimento.

4.3.5. Funciones para operar con las estadísticas

- **int reset_statisticks(void)**: resetea los arrays de estadísticas de cada una de las tareas del sistema.
- **int reset_global_statisticks(void)**: resetea la estructura global de estadísticas del sistema.
- **int reset_task_statisticks(struct task_struct *task)**: resetea las estadísticas de la tarea *task*.
- **void init_statisticks_struct(statisticks_struct *statistics)**: resetea el statisticks_struc *statistics*.

- **void update_statisticks_struct(statisticks_struct *statistics, unsigned int new_value)**: considera el nuevo valor *new_value* en las estructura estadística *statistics* incrementando el acumulador, actualizando el campo *last_value* y cambia si es preciso el valor de los campos *last_value_not_cero*, *min* y *max*.

- **void update_statisticks_struct_av(statisticks_struct *statistics, unsigned int new_value)**: actualiza con el valor new_value el buffer circular de mediciones de la estructura estadísticastatistics.

4.4. Actualización de mediciones dentro del kernel

4.4.1. Cuándo realizar las actualizaciones locales y globales

Una vez decididos los eventos a medir e insertados los experimentos resultantes de su traducción en el kernel el sistema debe activarlos para que empiezen a contar. Así mismo deben estar previamante reseteadas las estructuras de almacenamientos estadísticos de los valores.
Solo queda entonces, encontrar el momento en que el contenido de los contadores se tiene en cuenta en los resultados estadísticos.

Inicialmente se tanteó la siguiente situación:

- Las **mediciones locales** se actualizan en la función *context_switch()* (cada cambio de contexto).

  Es lógico pensar que la información de un proceso solo es necesario actualizarla cada vez que este pase por una de las cpu's. Así, basta con considerar en la función *context_switch()* la tarea current (la que está en ejecución en el procesador en el que se ejecuta la llamada) para la actualización de las estadísticas justo antes de que se haga el cambio de current por la nueva tarea que el planificador decida poner en ejecución.

- **Mediciones globales** se actualizan en la función *rebalance_tick()* (cada tick de reloj).
  Este punto determina un tick de reloj y es la mínima unidad de tiempo significativa que podemos considerar para actualizar las mediciones de todo el sistema.

Es necesario aclarar que un cambio de contexto se produce cada más tiempo que un tick de reloj, luego la frecuencia con que se llama a la función *context_switch()* es menor que la frecuencia con que se llama a *rebalance_tick()*.

La anterior decisión era acertada para símplemente hacer profiling midiendo un experimento global al sistema.
Sin embargo, si tenemos en cuenta que el algoritmo de la política de Calidad de Servicio necesita continuamente que se le suministren datos sobre la evolución de un proceso durante su ejecución, es muy poco útil que sólo se pueda disponer de esta cada vez que la tarea sale de la cpu (cambio de contexto).
Por ello se decidió que las mediciones locales deben actualizarse también con cada tick de reloj y tener así un seguimiento más continuo de la evolución del proceso estudiado en el algoritmo.

Finálmente, la actualización de ambas estadísticas queda fijada como se muestra:

- **Mediciones locales** se actualizan en la función *rebalance_tick()* cada *nticks* ticks de reloj.
- **Mediciones globales** se actualizan en la función *rebalance_tick()* cada tick de reloj.

Concrétamente se hará la actualización cada *nticks* ticks de reloj para que se permita graduar el nivel de detalle que se desea en el profiling. Este parámetro *nticks* es configurable desde usuario como se indicó en el capítulo 6.

4.4.2. Cómo realizar las actualizaciones locales y globales

- Mediciones globales dentro del kernel

  Para evitar redundancias deben hacerse siempre desde el procesador lógico cero (hecho que no tiene nada que ver con el procesador para el que está programado el evento).

Recordemos que los experimentos globales son aquellos de bajo nivel que se insertaron para ser contados relativos al sistema y no locales a un proceso y aquellos de alto nivel que se han insertado en el sistema como composiciones (sumas, restas, ratios etc) de los anteriores.
Su información debe ser por tanto almacenada en las estructuras creadas en el planificador para tal fin. Este es el campo *ll_statistics* de *global_prof* para las estadísticas de eventos de bajo nivel y el campo *hl_statistics* de la misma estructura *global_prof*.

- Mediciones locales dentro del kernel

La actualización que se ordena hacer en cada momento se hace sólo sobre la tarea que está corriendo en la cpu considerada.
Los experimentos locales de bajo nivel son aquellos que el usuario insertó para que midiesen (si la naturaleza del evento así lo permite como se contó en el apartado 2.2.1) eventos relativos a la traza de ejecución particular de un proceso.
Los experimentos locales de alto nivel serán composiciones (sumas, restas, ratios etc) de los anteriores, aunque no se ha impuesto que los dos operandos de estas composiciones tengan que ser de bajo nivel local.

La información estadística estará almacenada en el *task_t* de cada proceso, concretamente en el campo *prof* como se explicó en este mismo capítulo en 4.2.4.

El orden en que se hará la actualización es: primero las actualizaciones de los experimentos locales y luego los globales. En ambos casos primero se actualizarán las estadísticas de bajo nivel y luego las de alto.
Por como se hizo la inserción de los experimentos en los arrays de experimentos (ver apartado II) están resueltos todos los conflictos de dependencias, evitándose de esta manera ciclos en la composición de experimentos.
Se permite hacer composiciones cruzadas aunque los experimentos resultantes no tengan un significado lógico. Es decir, el usuario puede hacer un experimento de alto nivel global en función de experimentos de bajo nivel locales aunque como decimos, no tiene mucho interés. Normalmente los experimentos de alto nivel global estén en función de los experimentos de bajo nivel y alto nivel global y lo mismo para los locales, teniendo en cuenta esto el orden propuesto para las actualizaciones es correcto.

Las actualizaciones se desencadenan desde el código de *sched.c* con las llamadas a *do_count_local_events(this_cpu)* y la llamada a
do_count_global_events(this_cpu).

```
#ifdef CONFIG_SCHED_HTA
if ( (current->pid !=0) && (current->pid !=1) )
{
                        /*Accion de cuenta si procede*/
                        if (current->prof.ticks_counter==0)
                                        do_count_local_events(this_cpu);
                         /*Incremento del contador del control de cuenta*/
                        if(samples_info.nticks !=0)
                           current->prof.ticks_counter=(current->prof.ticks_counter+1)%samples_info.nticks;
                        (...)

                        if( this_cpu == 0 ) {
                                        do_count_global_events(this_cpu);
                                        if( __should_disable_smt( ) ) {
                                                activate_hta_thread = 1;
                                                wake_up_process( disable_ht_thread );
                                        }
                        }
#endif


void do_count_global_events(unsigned int this_cpu){
if (down_read_trylock(&sem_ht_exp)) {
        preempt_disable();
        doCount(&ht_exp,this_cpu,_COUNT_MODE_GLOBAL,
                global_prof.ll_statistics,
                global_prof.hl_statistics,
                NULL,
                ht_do_fill_buffer_global
        );
        preempt_enable_no_resched();
        up_read(&sem_ht_exp);
        }
}
void do_count_local_events(unsigned int this_cpu){
if (down_read_trylock(&sem_ht_exp)){
        preempt_disable();
        doCount(&ht_exp,this_cpu,_COUNT_MODE_LOCAL,
                current->prof.ll_statistics,
                current->prof.hl_statistics,
                NULL,
                ht_do_fill_buffer
        );
        preempt_enable_no_resched();
        ht_check_qos(this_cpu);
        up_read(&sem_ht_exp);
        }
}
```

En el caso de las mediciones locales además se incluye la llamada al algoritmo de Calidad de Servicio *ht_check_qos(this_cpu)* que se explicará más tarde.
En las anteriores funciones se observan las diferencias entre ambas actualizaciones fundamentalmente en los arrays de estadísticas que usan cada uno y en el llenado del triple buffer de estadísticas que hacen *ht_do_fill_buffer_global* para los globales y *ht_do_fill_buffer* para los locales. Este último es una llamada a una función que se pasa como parámetro a *doCount()* que se describe a continuación y que representa la acción que se desecadena tras hacer las mediciones.

```
void doCount(ht_experiment_t* ht_experiment,
                                unsigned int cpu,
                                count_mode mode,
                                statisticks_struct *ll_statistics,
                                statisticks_struct *hl_statistics,
                                void* data,
                                int (*action_function) (unsigned int,
                                        high_level_exp*,
```

```
                                        ht_experiment_t*,
                                        unsigned int,
                                        void*) )
{
                //Inicializamos los actuales
                ll_exps_set* cur_ll_set;
                hl_exps_set* cur_hl_set;
                unsigned int event_cpu=0;
                unsigned int i;
                unsigned int last_value;
                int keep_counting=1;
                if (mode ==_COUNT_MODE_LOCAL) {
                        cur_ll_set=&ht_experiment->ll_local_set;
                        cur_hl_set=&ht_experiment->hl_local_set;
                        }
                else { // _COUNT_MODE_GLOBAL
                                cur_ll_set=&ht_experiment->ll_global_set;
                                cur_hl_set=&ht_experiment->hl_global_set;
                }
                        for (i=0;i<cur_ll_set->size;i++) {
                        low_level_exp* lle=&cur_ll_set->array[i];
                        event_cpu=__getEventCPU(lle);
                                if ((mode ==_COUNT_MODE_LOCAL && cpu==event_cpu) ||
                                (mode ==_COUNT_MODE_GLOBAL && event_cpu==_IS_BOTH_CPU && cpu==0 ) ||
                                (mode ==_COUNT_MODE_ALWAYS))
                                {
                                        __stopCount(lle);
                                        __readCount(lle);
                                        __startCount(lle);
                                last_value=__getLastValue(lle);
                                update_statisticks_struct(&ll_statistics[i],last_value);
                                }
                        }
                        for (i=0;i<cur_hl_set->size && keep_counting ;i++){
                                high_level_exp* hle=&cur_hl_set->array[i];
                                compute_value(hle,ht_experiment,cpu);
                                last_value=hle->count;
                                update_statisticks_struct(&hl_statistics[i],last_value);
                                keep_counting=!action_function(i,hle,ht_experiment,cpu,data);
                        }
}
```

La llamada a *doCount* hace un recorrido actualizando el array de experimentos de bajo nivel si la cpu actual coincide con las especificada en la máscara de la configuración.
Para actualizar un experimento de bajo nivel, se para el correspondiente contador, se lee su valor y se reanuda la cuenta hasta la próxima medición.

Después, para cada uno de los experimentos de alto nivel global insertados se calcula el valor de los operandos y se hace el cálculo pertinente según sea suma, resta, multiplicación, división o ratio dentro de la función *compute_value(hle,ht_experiment,cpu)* que se explicó en este mismo capítulo. Además, dentro de esta función, en el caso de los ratio se considera el factor de escala que se haya configurado en el experimento de alto nivel correspondiente.

Una llamada a *doCount* como la mostrada no solo actualiza las estadísticas locales, sino que, actualiza también el buffer de resultados estadísticos que se usará como salida si el usuario quiere ver resultados de profiling mediante la interfaz mostrada en */proc* que se explicó en el apartado 6.
Ello se consigue pasando como parámetro un puntero a la función cuya acción se desea

ejecutar al terminar la actualización.

```
int  ht_do_fill_buffer (unsigned int evt_idx,
                                        high_level_exp* hle,
                                        ht_experiment_t* ht_experiment,
                                        unsigned int cpu,
                                        void* data
                                        )
{

                if (((current->pid == ppid_prof )|| ((current->parent)->pid == ppid_prof )))
                        fill_ht_buffer_prof_t(&buffer_prof[cpu],current->pid,evt_idx,hle->count);
                return 0;

}

int  ht_do_fill_buffer_global (unsigned int evt_idx,
                                        high_level_exp* hle,
                                        ht_experiment_t* ht_experiment,
                                        unsigned int cpu
                                        )
{
                if ( ppid_prof!=-1 )   //profiling activado
                                fill_ht_buffer_prof_t(&buffer_prof[cpu],-1,evt_idx,hle->count);
                return 0;
}
```

- **hle**: Puntero al último evento de alto nivel procesado.
- **ht_experiment**: Puntero al conjunto de experimentos insertados.
- **cpu**: Cpu en ejecución.

```
static inline void fill_ht_buffer_prof_t(ht_buffer_prof_t* ht_buf,unsigned int pid,
                                        unsigned int evt_idx, unsigned int last_count){

        if ( down_write_trylock(&ht_buf->sem) && ht_buf->oc_prof_buffer < PROF_BUF_SIZE)
                        {
                        //printk("New pid\n");
                                ht_buf->pid_arr[ht_buf->oc_prof_buffer]=pid;
                        //printk("New index\n");
                                ht_buf->index_event_arr[ht_buf->oc_prof_buffer]=evt_idx;
                        //printk("New last\n");
                                ht_buf->last_value_arr[ht_buf->oc_prof_buffer]=last_count;
                        //printk("New oc\n");
                                ht_buf->oc_prof_buffer +=1;
                        //printk("fin\n");
                        up_write(&ht_buf->sem);
                        }
}
```

En la función *fill_ht_buffer_prof_t()* se procede al llenado propiamente dicho del triple buffer que recordemos, será vaciado cada vez que se ordene volcar su contenido en el */proc* como se mostrará en el capítulo 5.8.

4.5. Tratamiento de los umbrales dinámicos

Para considerar que un proceso está sufriendo un pérdida de rendimiento nos basaremos en umbrales calculados dinámicamente.

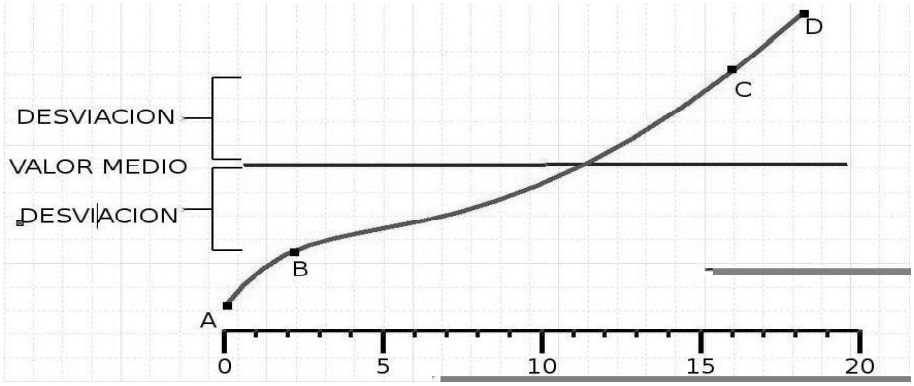


Figura 4.1: Diagráma de uso de los umbrales dinámicos.

Llamaremos *umbral dinámico* de un experimento para un proceso en un determinado momento al valor en media de las n últimas mediciones de un experimento relativas a dicho proceso. El valor n de mediciones consideradas corresponde al parámetro configurable (ver 4.2.4) *max_size* del buffer circular que se almacena en la estadísticas de cada proceso. Es por ello que se considera el tamaño de ventana del algoritmo.

Calculado este valor medio en un momento determinado, el umbral viene dado por:

[desviación, valor medio + desviación]

La medición realizada en este instante no estará incluida en la media mencionada y puede situarse en tres franjas. Veámoslo con la figura 4.1.

La curva mostrada corresponde a la trazada uniendo puntos que corresponden a mediciones tomadas.

- **Fuera del rango por abajo**
  Una medición correspondiente a alguno de los puntos en el intervalo [A,B] corresponde a una medición que se sale por debajo del umbral permitido.

- **Dentro del rango**
  Una medición correspondiente a alguno de los puntos en el intervalo [B,C] corresponde a una medición que está dentro de los valores permitidos del umbral.

- **Fuera del rango por arriba**
  Una medición correspondiente a alguno de los puntos en el intervalo [C,D] corresponde a una medición que se sale por encima del umbral permitido.

Lo restrictivo que se quiera ser respecto al ajuste al umbral se controla con el valor de desviación (ver 4.5).

Esta desviación es superior e inferior y permite que la medición oscile en el rango [desviación, valor medio + desviación] sin que se produzcan saltos frecuentes en el estado del procesador. Además permite ajustar el seguimiento del algoritmo a aquellos procesos cuya curva de evolución tenga numerosos picos de poca altura y anchura.

4.6. Mecanismos para activación /desactivación de hyperthreading

En el apartado 4.7.1 se verá como el estado del hyperthreading es sensible a la configuración del flag __ht_enbled. Ello significa que si se quiere activar basta con actualizar este flag con valor 1.
Por el contrario si se quiere desactivar no basta con ponerlo a cero puesto que la desactivación implica desencadenar acciones complejas como son un nuevo reparto de las colas de procesos activas, etc... La desactivación propiamente dicha se realizará en la función *rebalance_tick()* implementada en *sched.c* donde se llama a *__should_disable_smt(void)*.

```
int __should_disable_smt( void )
{
        if( __force_disable && __ht_enabled ) {
                __ht_enabled = 0;
                return 1;
        }
        else
        if( __sched_force_disable && __ht_enabled ) {
                __ht_enabled = 0;
                return 1;
        }
        else {
                return 0;
        }
}
```

```
int enable_smt(void)
{
         __sched_force_enable = 1;
        __sched_force_disable = 0;
        __ht_enabled = 1;
        ht_sr.state=HT_AUTO_ENABLED;
        printk("\nHT␣ENABLED\n");
        return 0;
}
```

```
int  disable_smt(void)
{
     __sched_force_enable = 0;
     __sched_force_disable = 1;
    ht_sr.state=HT_AUTO_DISABLED;
   printk("\nHT␣DISABLED\n");
   return 0;
 }
```

4.7. Calidad de Servicio: Algoritmo e Implementación

4.7.1. El planificador visto como una máquina de estados

La introducción de un comportamiento dinámico del sistema para activar y desactivar el hyperthreading, nos lleva a considerar el mismo como un autómata cuyo comportamiento se controla con un conjunto concreto de entradas.

El cuidadoso estudio de los valores que afectan al estado del planificador así como las acciones y comportamientso que debe manifestar según el estado al que haya sido llevado, hace que el sistema sea sensible a las pérdidas de rendimiento de tareas activas en el mismo, comportándose de manera inteligente para mejorar la calidad deservicio de dichas tareas.

A continuación se hará una breve descripción de los estados por los que puede pasar el planificador y de las entradas a las cuales es sensible.

Estados del planificador

La relación de los estados por los que el planificador se moverá quedan reflejados en la tabla 4.2. Para especificar las transiciones que se realizan entre ellos se han incluído los diagramas de estado de la figura 4.2.

Estas transiciones vienen explicadas en la tabla de verdad 4.1

Flags introducidos en el sistema

En el sistema hay cinco flags relevantes a la hora de controlar el estado del planificador.
El flag *__ht_enabled* indica el estado del hyperthreading mientras que el resto representan órdenes que se le dan al planificador sobre decisiones de activación o desactivación del funcionamiento de los dos procesadores lógicos.
Es importante destacar que los flags *__force_enable* y *__sched_force_enable* (y dualmente

Estado	**__force _enable __force _disable**	**__sched_force _disable __sched_force _enable**	**Estado siguiente**
HT_FORCE_DISABLED	X 0	X X	HT_FORCE_DISABLED
HT_FORCE_DISABLED	0 1	X X	HT_FORCE_ENABLED
HT_FORCE_ENABLED	0 X	X X	HT_FORCE_ENABLED
HT_FORCE_ENABLED	1 X	X X	HT_FORCE_DISABLED
HT_AUTO_ENABLED	0 0	0 X	HT_AUTO_ENABLED
HT_AUTO_ENABLED	0 0	1 X	HT_AUTO_DISABLED
HT_AUTO_DISABLED	0 0	X 0	HT_AUTO_DISABLED
HT_AUTO_DISABLED	0 0	X 1	HT_AUTO_ENABLED
HT_AUTO_ENABLED	1 0	X X	HT_FORCE_DISABLED
HT_AUTO_ENABLED	0 1	X X	HT_FORCE_ENABLED
HT_AUTO_DISABLED	1 0	X X	HT_FORCE_DISABLED
HT_AUTO_DISABLED	0 1	X X	HT_FORCE_ENABLED
HT_WAITING _TIMESLICE	0 1	X X	HT_FORCE_ENABLED
HT_WAITING _TIMESLICE	1 0	X X	HT_FORCE_DISABLED
HT_WAITING _TIMESLICE	0 0	0 0	HT_WAITING _TIMESLICE
HT_WAITING _TIMESLICE	0 0	0 1	HT_AUTO_ENABLED
HT_WAITING _TIMESLICE	0 0	0 1	HT_AUTO_DISABLED

Cuadro 4.1: Tabla de verdad para la transición de los estados del planificador

sus correspondientes *__force_disable* y *__sched_force_disable*) funcionan de manera similar. Esta aparente redundancia se debe a la separación de la información perteneciente al algoritmo de calidad de servicio y a la información de los deseos del usuario (este puede por ejemplo forzar la activación del hyperthreading).
Lógicamente, los flags *__force_enable* y *__force_disable* tienen prioridad ante *__sched_force_enable*

Estado	Comportamiento del planificador
HT_FORCE _ENABLED	El hyperthreading está activado permanentemente. no se hace ninguna de las consideraciones relacionadas con la calidad de servicio anteriormente explicada.
HT_FORCE _DISABLED	El hyperthreading está desactivado permanentemente. no se hace ninguna de las consideraciones relacionadas con la calidad de servicio anteriormente explicada.
HT_AUTO _ENABLED	El planificador está en modo *auto*, es decir, está activando y desactivando en caliente el hyperthreading en función de consideraciones relativas a la calidad de servicio. Actualmente, el planificador ha detectado siguiendo el algoritmo de calidad de servicio que debe estar activado el hyperthreading.
HT_AUTO _DISABLED	El planificador está en modo *auto*, es decir, está activando y desactivando en caliente el hyperthreading en función de consideraciones relativas a la calidad de servicio. Actualmente, el planificador ha detectado siguiendo el algoritmo de calidad de servicio que debe estar desactivado el hyperthreading.
HT_WAITING _TIMESLICE	El planificador está en modo *auto*, es decir, está activando y desactivando en caliente el hyperthreading en función de consideraciones relativas a la calidad de servicio. Actualmente, el planificador ha detectado que una tarea ha sufrido una violación de su rendimiento y desactivando el hyperthreading dicha tarea ha respondido satisfactoriamente volviendo a su comportamiento deseado. El planificador mantiene el hyhyperthreading desactivado hasta que esta tarea agote el timeslice que le fué otorgado. Cuando este finalice se activará de nuevo el hyperthreading.

Cuadro 4.2: Tabla de Estados del Planificador

y *__sched_force_disable* porque corresponden a deseos del usuario e inhiben al planificador de toda decisión relativa a activación o desactivación del hyperthreading en función de rendimiento local.

A continuación se detalla el significado de cada flag.

- **int __ht_enabled**: flag que indica el estado de hyperthreading en el sistema. Tomará valor 1 si este está activado y 0 en caso de no estarlo.

- **int __force_enable**: si su valor es 1, significa que se ha ordenado activar permanetemente hyperthreading. Este valor solo puede ser modificado desde el exterior (usuario).

- **int __force_disable**: dualmente al anterior, si su valor es 1, significa que se ha ordenado desactivar permanéntemente hyperthreading.

- **int __sched_force_enable**: si su valor es 1, el algoritmo de calidad de servicio ha tomado la determinación de activar el hyperthreading. Este valor no se puede modificar exteriórmente, sólo el planificador en función de la respuesta que de la terea a la cual se está haciendo la optimización de rendimiento.

- **int __sched_force_disable**: similar al anterior pero desactivando el hyperthreading.

Factores que provocan el cambio de estado del planificador

El tránsito del sistema de un estado a otro de los anteriórmente enumerados puede deberse a diversos factores.
Para hacer este análisis es necesario diferenciar dos situaciones distintas:

- Si el sistema no está funcionando en modo *auto* (en modo calidad de servicio) estará en el estado *HT_FORCE_DISABLED* ó *HT_FORCE_ENABLED* según haya decidido el usuario puesto que este funcionalidad se ha hecho configurable.

- Si el sistema sí está funcionando en modo *auto* estará moviéndose entre los estados *HT_AUTO_ENABLED*, *HT_AUTO_DISABLED* y *HT_WAITING_TIMESLICE* según decida el algoritmo de calidad de servicio.

El tránsito de un modo de funcionamiento a otro por deseos del usuario se hace mediante las pertinentes modificaciones en el fichero */proc* como se indicó en el capítulo 5.8.

Por tanto el primer factor que afecta al estado en el que se haya el planificador es el modo de funcionamiento que escoja el usuario: modo auto o modo no auto (ordenando poner el hyperthreading en enabled o disabled).

Si estamos en **modo auto**, el sistema seguirá el tránsito de estados que dicte el **algoritmo de calidad de servicio** aplicado a la situación determinada de las tareas en el sistema en un momento dado. Es decir, en función del parámetro respecto al cual se oriente la calidad de servicio el hyperthreading se activará y desactivará de forma automática por parte del sistema.

A grandes rasgos, el algortimo de Calidad de Servicio es el siguiente. En la elección **en modo auto**, se llevará inicialmente al sistema al estado *HT_AUTO_ENABLED*. Cuando una tarea de uno de los dos procesadores sea más prioritaria que la que está en el otro procesador estaremos en una situación en la que puede producirse una pérdida de rendimiento. Si se detecta que el valor medido en ese momento del experimento para el cual se orientó la calidad en la tarea más prioritaria tiene un valor que se sale del rango permitido por el umbral impuesto se habrá producido una violación del rendimiento de dicha tarea. La decisión del planificador será desactivar el hyperthreading para que esa tarea goze de más los recursos del sistema en exclusividad y no compartiéndolos con otra trarea en la otra cpu lógica. Se habrá pasado entonces al estado *HT_AUTO_DISABLED*.

Si pasado una cantidad de ticks mínima el valor de dicho experimento medido en la tarea violada no presenta mejoras, era porque la disminución anteriormente detectada formaba parte del comportamiento natural de dicho proceso, y por tanto, no es necesario mantener el hyperthreading desactivado. Se activará devolviéndo al sistema al estado *HT_AUTO_ENABLED* y volviendo a empezar el ciclo del autómata.

Si por el contrario, el proceso detecta una mejora en su rendimiento es porque le benefició el desactivar el hyperthreading. Se tomará la decisión de dejarlo desactivado un tiempo prudencial para dar lugar a que la tarea se sirva con más rendimiento local. Se lleva para ello al planificador al estado *HT_WAITING_TIMESLICE* y en él permanecerá hasta que termine el timeslice (ver 10) que le fué asignado. Entonces se volverá a activar el hyperthreading devolviendo al planificador al estado *HT_AUTO_ENABLED* y volviendo a empezar otro nuevo ciclo del autómata.

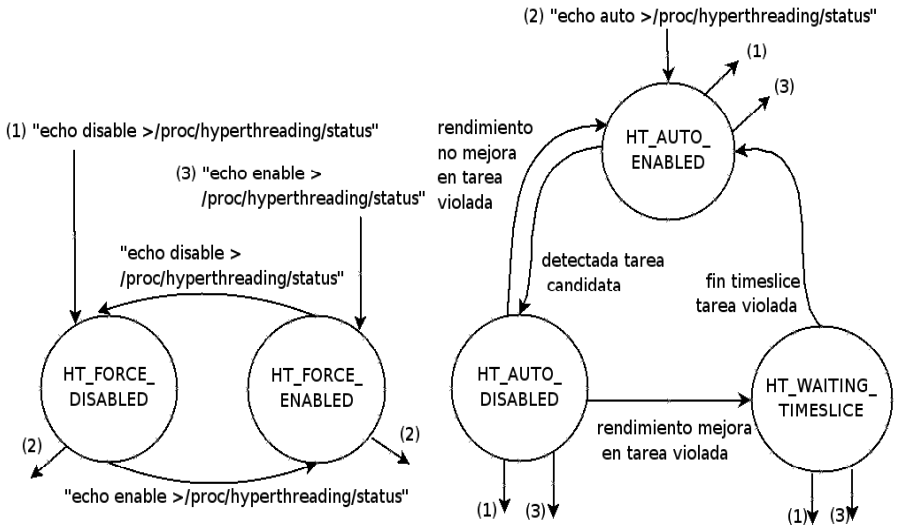


Figura 4.2: Diagráma de transición de estados del planificador.

4.7.2. Detección de pérdida de rendimiento y recuperación

Para ofrecer localmente calidad de servicio sin perjudicar al rendimiento del resto de tareas en el sistema, es necesario hacer un cuidadoso seguimiento de la tarea cuya calidad se quiere mejorar.
Explicaremos a continuación el algortimo que sigue el planificador al actuar en modo auto para ofrecer calidad de servicio.

Tres puntos son clave en este algoritmo: la detección de la tarea cuyo rendimiento está siendo perjudicado, el estudio de la evolución de dicha tarea al desencadenarse las pertinentes acciones por parte del planificador y el compromiso con el resto de tareas para asegurar que la mejora en rendimiento de una tarea no conlleva la pérdida significativa del rendimiento del resto y en general, global del sistema.

- **Detección de un proceso con pérdida de rendimiento con hyperthreading activado.**

  El chequeo de calidad de servicio se hace cada tantos ticks de reloj como se indiquen en *nticks* para todas las tareas (como la toma de medidas) siempre que el planificador esté funcionando en modo auto y que ya no se este ofreciendo la calidad de servicio a otra tarea. Para que una tarea sea identificada como violada en rendimiento ha de cumplir las condiciones siguientes:

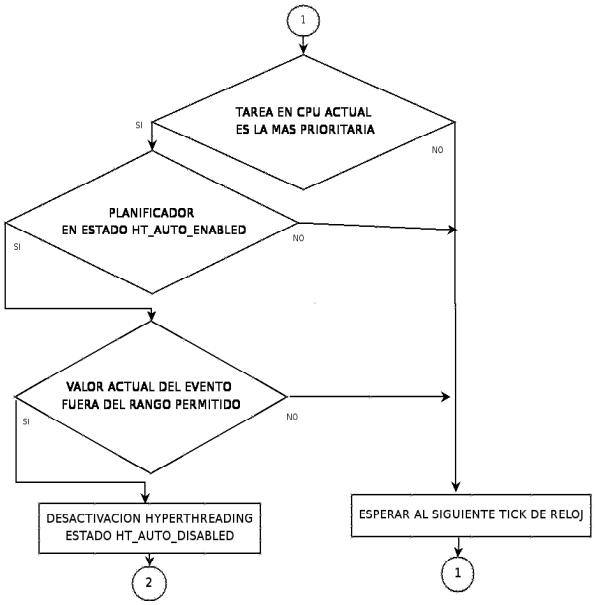


Figura 4.3: Diagráma de flujo algoritmo qos. Parte 1.

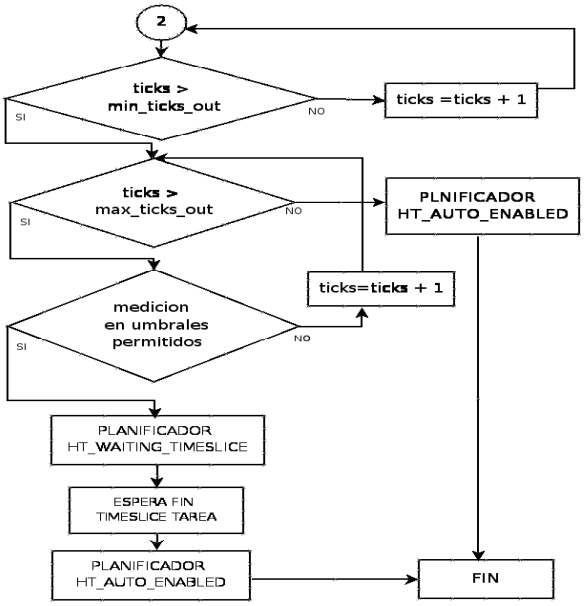


Figura 4.4: Diagráma deflujo algoritmo qos. Parte 2.

1. **Ser la tarea más prioritaria que está en el estado *TASK_RUNNING"* de las dos cpu's**. Es necesaro inspeccionar la prioridad de las dos tareas que están en cada procesador lógico. Basta para ello consultar la prioridad estática de los procesos de cada cola. En casos como el nuestro, en el que se tienen dos procesadores lógicos, esta prioridad está en el campo *static_prio* de la tarea *current* para el caso de la cpu en la que se hace la comprobación y en el campo *static_prio* de la tarea *current* de la otra cpu que viene dada porla macro *cpu_rq(other_processor)*.
   La comprobación de que la tarea en la cpu vigente es la más prioritaria se hace en la función *int me_more_priority(unsigned int my_cpu)*.
2. **Estar el planificador en el estado HT_AUTO_ENABLED**. Puesto que la oferta de calidad de servicio solo tiene sentido en el contexto de competencia de recursos por parte de dos tareas, la detección de una tarea con rendimiento en decremento ha de ser realizada con el hyperthreading activado.
   Por tanto, el planificador ha de estar en el estado HT_AUTO_ENABLED.
3. **Detectar una pérdida de rendimiento**. Para que la tarea se considere violada en rendimiento ha de observarse un decreciemiento (en el caso eventos cuyo decrecimiento es negativo como por ejemplo el *ipc*) considerable en el valor del evento al cual se orienta la calidad de servicio. En nuestro caso se orientó dicha calidad al evento que media el ipc de un proceso, por ello consideraremos que el proceso manifiesta una pérdida de rendimiento si el valor de este decrece. Ello nos lleva a clasificar la tarea respecto un umbral dinámico. La explicación de cálculo y clasificación de tareas respecto umbrales dinámicos se especificó en el apartado 4.5.
   Si la tarea con su última medición del evento en cuestión presenta la situación *HT_INTERVAL_DOWN* respecto sus umbrales dinámicos entonces presenta una pérdida de rendimiento considerable.

Si una tarea cumple la primera y tercera condición estando el planificador en el estado explicado en la segunda condición se habrá detectado una situación en la que es necesario aplicar el algoritmo de Calidad de Servicio.

- **Acciones desencadenadas: desactivación del hyperthreading y conservación del contexto en que se produjo la desactivación**.

Puesto que para la tarea detectada parece ser pernicioso la activación del hyperthreading ha de procederse a desactivar el mismo llevándo al planificador al estado HT_AUTO_DISABLED.
Además, para poder hacer un seguimiento del comportamiento de la misma y

detectar si se han producido o no mejoras en ella ha de conservarse cierta información. El planificador debe conservar el identificador de proceso que la sufrió y cuál es el valor deseado que la medición del evento debe alcanzar.
Así, cada vez que vuelva a pasar la tarea por la cpu 0 (única activa con el hyperthreading desactivado) se podrá chequear si ha mejorado.

La estructura de datos usada para mantener esta información en el planificador se explicó en el apartado 4.2.1.

También, ha de hacerse un reseteo del contador de ticks globales que pasan desde que se produjo la desactivación por motivos de rendimiento.
En el momento en que se detectó la violación se hace:

```
if ( ( ht_sr.state==HT_AUTO_ENABLED) &&
     (current_situation==HT_INTERVAL_DOWN) &&
      me_more_priority(cpu))
        {
                ht_sr.idx_evt=evt_idx;
                ht_sr.desired_average=cur_statistics->average ;
                ht_sr.qos_ticks_counter=0;
                ht_sr.pid_violated=current->pid;
                disable_smt();

        }
```

- **Seguimiento del comportamiento de un proceso con calidad violada**.

Desde el momento de la desactivación, el procesos tiene una oportunidad para remontar su comportamiento.
Puesto que esta mejora no es inmediata, ha de darse a la tarea un tiempo inical mínimo en el que los valores que tome para ella el evento medido no son considerados. Este tiempo mínimo se cuantifica en ticks de reloj y corresponde con el valor de la variable *min_ticks_out* .
Pasado este tiempo prudencial, se procede a inspeccionar la reacción de la tarea frente a la desactivación.

1. **La tarea mejora su rendimiento: el otro proceso le estaba perjudicando**.
   Si pasado este intervalo mínimo de ticks, la tarea detecta que el valor medido del evento está dentro de los valores permitidos entonces se ha mejorado el rendimiento de la misma cumpliéndose los objetivos del algoritmo de calidad de servicio.
   Por ello se pone al planificador en el estado HT_WAITING_TIMESLICE consistente en esperar a que el proceso agote el timeslice que le fué aplicado de acuerdo con su prioridad. Así, se favorecera el curso de este proceso con la desactivación del hyperthreading.

2. **La tarea no mejora su rendimiento: el otro proceso no era el causante de su comportamiento**.
   Se ha dejado a la tarea una cantidad considerable de ticks de reloj (desde *min_ticks_out* hasta *max_ticks_out*)y no se ha observado ninguna mejora en su comportamiento.
   Esto indica que la desactivación del hyperthreading no fué una decisión correcta porque la disminución observada en el momento de la detección era debida al comportamiento natural del proceso y no a competencia de recursos con el otro proceso activo.
   Se toma por tanto la decisión de activar el hyperthreading de nuevo desechando la decisión de mejora de calidad de servicio porque el proceso considerado no puede mejorar su rendimiento.

El estudio del valor óptimo de las varibles *min_ticks_out* y *max_ticks_out* se refleja en el apartado 7.2. Es importante dar con los valores óptimos del rango que acota el intervalo relevante en el estudio de calidad de servicio para un proceso. Puesto que estos valores se han hecho configurables dinámicamente se ha sometido al sistema pruebas hasta dar con los vlores óptimos de los mismos.

- **Recuperación de un proceso con rendimiento violado**.

Se ha dejado el hyperthreading desactivado favoreciendo al proceso que fué violado.
Durante este tiempo, se mantuvo al planificador en el estado
HT_WAITING_TIMESLICE. Es necesario por tanto inspeccionar el valor del timeslice en cada ráfaga de ejecución de cpu de la tarea para detectar el momento en que se agote y habilitar el hyperthreading. Además, tenemos en cuenta que el proceso puede terminar una ráfaga de cpu con el valor de su timeslice sin agotar y salir porque haya terminado.

- **Compromiso con el rendimiento del sistema**.

La desactivación del hyperthreading ha de hacerse de manera controlada puesto que una desactivación desatinada puede producir pérdidas considerables en el rendimiento global del sistema.
Es por ello que solo se mantiene desactivado para favorecer a una tarea el tiempo mínimo necesario, esto es, el que le haya sido asignado en su timeslice.

```
static inline int enable_if_waiting_timeslice(pid_t pid){
int success;

                        CRITICAL_DOWN_WRITE_HT_SR;
                        if(ht_sr.state==HT_WAITING_TIMESLICE){
                                        printk("Enabling␣HT:␣timeslice␣expired␣for␣%i\n",pid);
                                        enable_smt();
                                        success=1;
                        }
                        else {
                                success=0;
                        }
                        CRITICAL_UP_WRITE_HT_SR;

        return success;
}
```

Capítulo 5

Concurrencia en el kernel de linux

En los sistemas operativos actuales existen numerosas fuentes de concurrencia. Esto se debe En kernels modernos de la distribución de linux hay gran cantidad de concurrencia y en consecuencia gran número posibles condiciones de carrera, inaniciones y deadlocks. Esto es debido a que a la vez corren en la máquina distintos procesos que pueden acceder al mismo código de incalculables maneras, esto se complica aún más si tenemos en cuenta que en sistemas SMP el código puede ejecutarse simultaneamente en varios procesadores, lo cual fue introducido en la versión 2.0 del kernel sin olvidar que con la introducción de expropiación en el kernel (desde la versión 2.6 de este), un proceso que se está ejecutando puede perder el control de la CPU (ser expropiado) en cualquier momento por otro más prioritario.

En este punto trataremos la importancia de la concurrencia y la sincronización en el kernel de linux, haremos un recorrido por las principales técnicas y explicaremos que tipos de estructuras se manejan. Además desarrollaremos las dificultades que hemos encontrado y qué motivos nos han llevado a tomar las decisiones correspondientes respecto a sincronización y concurrencia.

5.1. Introducción a la concurrencia

Las condiciones de carrera surgen por el acceso concurrente a recursos compartidos. Si dos hilos de ejecución trabajan con la misma estructura de datos hemos de tener mucho cuidado. La solución parece trivial, evitemos las variables globales y hagamos que cada hilo de ejecución trabaje solo con sus variables locales. Sin embargo, esto no es una solución suficiente. En el contexto de nuestro proyecto hay casos como las estructuras donde se almacenan los experimentos que se usan para hacer profiling y calidad de servicio, que han de ser globales. Además hemos de tener en cuenta que no sólo las variables globales son la única manera de compartir datos. En cualquier sitio del código del sistema en el que se pasa un puntero a alguna parte del kernel se está potencialmente creando una nueva situación de compartición. Por tanto, cada vez

que haya datos compartidos por distintos hilos de ejecución existe la posibilidad de que uno de ellos encuentre datos inconsistentes. Por tanto se debe controlar el acceso explícitamente.

Pongamos un ejemplo simple que ilustre la sutileza del problema. Imaginemos dos hilos de ejecución en el que cada uno ejecuta el siguiente código:

```
i++;
```

Supongamos que la variable i es global a los dos hilos y que está inicializada a 0. Un posible comportamiento sería el de la tabla 5.1.

Hilo 1	Hilo 2	Valor de i
	-	0
leer i		0
incrementar i		0
escribir i		1
	leer i	1
	incrementar i	1
	escribir i	2

Cuadro 5.1: Ejemplo de ejecución concurrente de dos hilos (1)

En el ejemplo anterior la variable i acaba valiendo 2, pero veamos que pasa en el siguiente caso que se muestra en la tabla 5.2:

Hilo 1	Hilo 2	Valor de i
	-	0
leer i		0
	leer i	0
	incrementar i	0
incrementar i		0
escribir i		1
	escribir i	1

Cuadro 5.2: Ejemplo de ejecución concurrente de dos hilos (2)

Ahora la variable toma el valor 1. Si en un caso tan sencillo podemos tener indeterminismo, uno puede imaginar la complejidad para controlar el acceso concurrente y la sincronización a lo largo de todo el kernel de linux.

Afortunádamente el kernel nos facilita una serie de herramientas para tratar la concurrencia. Se listan las mismas y sus descripciones en el cuadro 5.3:

Técnica	**Descripción**	**Alcance**
Operaciones atómicas	Instrucción para leer-modificar-escribir un contador atómicamente	Todas las CPUs
Barreras de memoria	Evita la reordenación de instrucciones	CPU local
Spinlock	Cerrojo con espera activa	Todas las CPUs
Semáforos	Cerrojo con espera bloqueante	Todas las CPUs
Deshabilitación de expropiación	Prohibe la expropiación de procesos	Todas las CPUs
Deshabilitación de interrupciones locales	Prohibe interrupciones en una CPU	CPU local
Deshabilitación de interrupciones globales	Prohibe interrupciones en todas las CPUs	Todas las CPUs

Cuadro 5.3: Herramientas para la concurrencia

5.2. Operaciones atómicas

No podemos suponer que todas las modificaciones de una variable se realicen de manera atómica. Una simple asígnación puede traducirse como un conjunto de instrucciones máquina, lo que potencialmente puede llevar a condiciones de carrera. Sin embargo no es posible que 2 operaciones atómicas operen sobre una misma variable de manera simultánea. Esta es la base de este tipo de sincronización.

El kernel dispone del tipo atomic_t (definido en `<asm/atomic.h>`)que almacena un valor entero sobre el cual se definen una serie de operaciones atómicas utilizando instrucciones máquina dependientes de las arquitecturas. Hemos de tener en cuenta que estas operaciones sólo son efectivas cuando no necesitamos operar con distintas variables atomic_t, que interaccionan entre sí. Pese a ser código que se ejecuta de manera muy eficiente por su simplicidad; esta misma simplicidad hace que sean insuficientes para muchas de las situaciones con las que nos hemos tenido que enfrentar.

Función	**Descripción**
atomic_read(v)	devuelve *v
atomic_set(v,i)	pone *v a i
atomic_add(i,v)	Suma i a *v
atomic_sub(i,v)	Resta i de *v
atomic_sub_and_test(i, v)	Resta i de *v y devuelve 1 si el resultado es cero; de lo contrario devuelve 0.
atomic_inc(v)	Suma 1 a *v
atomic_dec(v)	Resta 1 de *v
atomic_dec_and_test(v)	Resta 1 de *v y devuelve 1 si el resultado es cero; de lo contrario devuelve 0.
atomic_inc_and_test(v)	Suma 1 a *v y devuelve 1 si el resultado es cero; de lo contrario devuelve 0.
atomic_add_negative(i, v)	Suma 1 a *v y devuelve 1 si el resultado es negativo; de lo contrario devuelve 0.

Cuadro 5.4: Operaciones sobre variables atomic_t

Aparte de estas operaciones atómicas para enteros, el kernel también proporciona operaciones que operan a nivel de bit de manera atómica (definidos en `<asm/bitops.h>`).

En lugar de trabajar como antes con un argumento de tipo atomic_t, trabajan con punteros genéricos.

Función	**Descripción**
void set_bit(int nr, void *addr)	Activa el bit número nr empezando desde addr
void clear_bit(int nr, void *addr)	Desactiva el bit número nr empezando desde addr
void change_bit(int nr, void *addr)	Cambia el valor del bit número nr empezando desde addr
int test_and_set_bit(int nr, void *addr)	Activa el bit número nr empezando desde addr y devuelve el valor anterior
int test_and_clear_bit(int nr, void *addr)	Desactiva el bit número nr empezando desde addr y devuelve el valor anterior
int test_and_change_bit(int nr, void *addr)	Cambia el valor del bit número nr empezando desde addr y devuelve el valor anterior
int test_bit(int nr, void *addr)	Devuelve el valor del bit número nr empezando desde addr

Cuadro 5.5: Operaciones a nivel de bit de manera atómica

En un primer momento se intentó gestionar la concurrencia utilizando una solución basada en el variables de tipo atomic_t. Debido a su bajo nivel de abstracción y a que hay bastantes estructuras de datos compartidas no nos son útiles; esta estrategia no soluciona bien todos los problemas de concurrencia. En su lugar tuvimos que usar algo que nos proporcionara una funcionalidad y potencia mayores.

5.3. Barreras de Memoria

Debido a la situación actual de las computadoras no podemos suponer que las instrucciones que hayamos escrito en el código fuente se van a ejecutar en el procesador en el mismo orden en que las hemos escrito, las optimizaciones de los compiladores optimizan el uso de los registros, reordenan los accesos a memoria, etc...e incluso el mismo procesador realiza un reordenamiento de las microinstrucciones en función de sus compatibilidades y dependencias. El efecto que produce una barrera consiste en garantizar que todas las operaciones colocadas antes de la primitiva de sincronización son terminadas antes de empezar con las que vienen después de esta.

Es muy importante destacar que las barreras de memoria afectan muy negativamente al rendimiento, por eso no las hemos utilizado en ningún momento, ya que gran parte del código de nuestro proyecto se ejecuta en cada tick de reloj, si incluyéramos barreras el rendimiento se vería seriamente afectado.

Macro	**Descripción**
mb( )	Barrera de memoria para P y MP
rmb( )	Barrera de memoria de lectura para P y MP
wmb( )	Barrera de memoria de escritura para P y MP
smp_mb( )	Barrera de memoria sólo para MP
smp_rmb( )	Barrera de memoria de lectura sólo para MP
smp_wmb( )	Barrera de memoria de escritura sólo para MP

Cuadro 5.6: Operaciones sobre barreras de memoria

P = Monoprocesador

MP = MultiProcesador

5.4. Spinlocks

Se trata de un tipo de cerrojo especialmente diseñado para trabajar en entornos multiprocesador (o uniprocesador de comportamiento expropiativo), del que solo puede tener el control un hilo de ejecución.

Si al intentar tomar el control de un cerrojo este se encuentra abierto símplemente lo toma y continúa su ejecución, sin embargo si se encuentra cerrado ejecuta un bucle de espera activa hasta que es liberado, aunque suene algo excesivo la espera activa frente a la espera bloqueante, en estructuras de kernel es muy conveniente en ocasiones, ya que hay recursos que símplemente se bloquean fracciones de tiempo despreciables y se pierde más tiempo si se hace que el proceso se suspenda y que más adelante cuando reciba la señal correspondiente tenga que volver a intentar conseguirlo (espera bloqueante). Para realizar la espera bloqueante se pueden utilizar semáforos que también vienen implementados en el kernel como se explica en la siguiente sección.

Se utilizan por tanto para controla el acceso a las regiones críticas del código y además procurando respetar la naturaleza del spinlock, intentar minimizar el tiempo que se sostiene el bloqueo, ya que si hay más procesos intentando tomar el control se producirá bastante sobrecarga. Además a diferencia de los semáforos se pueden usar en trozos de código que no pueden dormir, como son los manejadores de interrupciones y si son corréctamente usados proporcionan un rendimiento más alto que los semáforos en general.

El tipo spinloc_t así como sus operaciones se haya definido en `<asm/spinlock.h>`.

En nuestro código utilizamos spinlocks para controlar el acceso al estado del hyperthreading cuando estamos haciendo calidad de servicio, las variables atómicas se nos quedan cortas en funcionalidad y los semáforos tienen el problema de que si no se adquieren el proceso se suspende, por tanto optamos por utilizar spinlocks en este caso, además el trozo de código que actualiza el estado del hyperthreading es mínimo por lo que no se introduce demasiada sobrecarga.

Función	**Descripción**
spin_lock_init()	Inicializa el spinlock_t dado
spin_lock( )	Espera hasta que el cerrojo esté liberado y luego lo bloquea
spin_trylock()	Intenta adquirir el cerrojo y si no está disponible devuelve un valor distinto de cero
spin_is_locked()	Devuelve un valor distinto de cero si el cerrojo ha sido adquirido por otro proceso, en otro caso devuelve cero
spin_unlock( )	Libera el cerrojo
spin_unlock_wait( )	Espera a que el cerrojo sea liberado
spin_is_locked( )	Devuelve 0 si el cerrojo se encuentra liberado, en otro caso devuelve 1
spin_trylock( )	Si está desbloqueado, bloquea el cerrojo y devuelve 1 si no devuelve 0.
spin_lock_irq()	Deshabilita las interrupciones y hace un spin_lock( )
spin_lock_irqsave()	Salva el estado de las interrupciones locales, deshabilita las interrupciones y hace un spin_lock( )
spin_unlock_irq()	Libera un cerrojo y habilita las interrupciones locales
spin_unlock_irqrestore()	Libera un cerrojo, habilita las interrupciones locales y vuelve al estado anterior

Cuadro 5.7: Operaciones sobre variables spinloc_t

5.4.1. Spinlocks de lectura escritura

El acceso concurrente a las estructuras de datos causa problemas sobre todo en la escritura, es decir si hay varios procesos intentando escribir a la vez, sin embargo si solo tenemos procesos que leen y ninguno que escribe no tenemos problema con la coherencia de la información. Estos spinlocks hacen que varios lectores pueden adquirir el control a la vez, sin embargo sólo permiten un escritor sin ningún lector.De esta manera si hay pocos escritores y muchos lectores se optimiza el acceso.

Función	Descripción
rw_lock_init()	Inicializa el rwlock_t dado
rw_is_locked()	Devuelve un valor distinto de cero si el cerrojo ha sido adquirido por algún otro proceso y cero en caso contrario
read_lock()	Adquiere el cerrojo para lectura
read_lock_irq()	Deshabilita las interrupciones locales y adquiere el cerrojo para lectura
read_lock_irqsave()	Salva el estado local de las interrupciones locales, deshabilita las interrupciones locales y adquiere el cerrojo para
lectura read_unlock()	Libera el cerrojo para lectura
read_unlock_irq()	Libera el cerrojo para lectura y habilita las interrupciones locales
read_unlock_irqrestore()	Libera el cerrojo para lectura ,habilita las interrupciones locales y restaura las interrupciones locales al estado anterior
write_lock()	Adquiere el cerrojo para escritura
write_lock_irq()	Deshabilita las interrupciones locales y adquiere el cerrojo para escritura
write_lock_irqsave()	Salva el estado local de las interrupciones locales, deshabilita las interrupciones locales y adquiere el cerrojo para escritura
write_unlock()	Libera el cerrojo para escritura
write_unlock_irq()	Libera el cerrojo para escritura y habilita las interrupciones locales
write_unlock_irqrestore()	Libera el cerrojo para escritura ,habilita las interrupciones locales y restaura las interrupciones locales al estado anterior

Cuadro 5.8: Operaciones sobre cerrojos

5.5. Semáforos

Otra herramienta de sincrozación muy importante son los semáforos. Cuando un proceso intenta obtener un recurso ocupado protegido por un semáforo del kernel, a diferencia de hacer espera activa como con los spinlocks el proceso correspondiente es suspendido hasta que el recurso sea liberado, momento en el cual el proceso volverá al estado RUNNABLE.

Los semáforos son estructuras muy típicas, realmente no son más que un entero almacenado y un par de funciones (P y V) que intentan incrementar o decrementar su valor. Si un proceso desea entrar a su sección crítica llama a la función P, si el valor almacenador es mayor que 0 lo decrementa, en otro caso el proceso espera (es suspendido) hasta que alguien libere el semáforo. El semáforo se libera llamando a V, que incrementa el valor del semáforo y si es necesario despierta al proceso que está esperando.

El tipo semaphore así como las funciones referentes a él se hayan en `<asm/semaphore.h>`

Función	**Descripción**
void sema_init(struct semaphore *sem, int val)	Inicializa el semáforo con val como valor inicial
void init_MUTEX(struct semaphore *sem)	Inicializa el semáforo abierto
void init_MUTEX_LOCKED(struct semaphore *sem)	Inicializa el semáforo cerrado
void down(struct semaphore *sem)	Intenta adquirir el semáforo y si no lo obtiene el semáforo se suspende
int down_interruptible(struct semaphore *sem)	Igual que down pero la operación es ininterrumpible
int down_trylock(struct semaphore *sem);	Intenta adquirir un semáforo, pero con la peculiaridad de que nunca duerme, si el semáforo no está disponible devuelve un valor distinto de cero pero vuelve inmediatamente
void up(struct semaphore *sem)	Libera el semáforo

Cuadro 5.9: Operaciones sobre variables semáforos *semaphore*

En muchas ocasiones se usan semáforos como cerrojos, símplemente para garantizar la exclusión mutua entre dos procesos (en cada momento se permite que sólo un proceso esté dentro del semáforo), para hacerlo de manera más cómoda se dispone de una serie de funciones y macros adicionales: DECLARE_MUTEX(name); DECLARE_MUTEX_LOCKED(name); Que dan como resultado un semáforo inicializado a 1 (abierto)con DECLARE_MUTEX o inicializado a 0 (cerrado) con DECLARE_MUTEX_LOCKED.

Para controlar el acceso a las estructuras de nuestro código hemos tenido que usar semáforos, esto es así porque hay llamadas al sistema que las modifican. En este caso

sí que nos interesa que el proceso que lleva a cabo esta modificación no haga espera activa, sino que quede suspendido hasta que pueda realizar la modificación. Además los spin_locks sólo permiten estar en la sección crítica a un proceso, mientras que con un semáforo se generaliza a n procesos, en el procesador sobre el que se hicieron las pruebas sólo había 2 procesadores lógicos, pero si tuvieramos que generalizarlo para n procesadores podríamos hacerlo manteniendo los mismos semáforos mientras que con spin locks no. También hay que tener en que cuenta que en el resto de zonas en las que se accede a estas estructuras se utilizan las llamadas con trylock que devuelven un valor informando de si se dispone del semáforo o no, esto optimiza mucho el rendimiento ya que no se manda al proceso a dormir cada vez que no obtiene el semáforo.

5.5.1. Semáforos de lectura escritura

Cómo se ha comentado anteriormente se han utilizado semáforos en la implementación del proyecto, sin embargo no se ha utilizado la implementación básica, se ha utilizado un tipo especial (rwsem o semáforos de lectura escritura, que aparecen definidos en `<linux/rwsem.h>`) que permite que haya un escritor y varios lectores (de manera similar a lo ya explicado sobre los spinlocks de lectura escritura) debido a que la naturaleza de las estructuras de nuestro código se caracteriza por realizar pocas escrituras y muchos accesos de lectura. Por ejemplo en el caso de la estructura de datos que maneja los experimentos solo se escribe cuando se modifican a través de la llamada al sistema, sin embargo el acceso en lectura si se está haciendo profiling se hace cada cierto número de ticks de reloj configurable a través de /proc.

Función	**Descripción**
void init_rwsem(struct rw_semaphore *sem)	Inicializa el semáforo de lectura escritura
void down_read(struct semaphore *sem)	Intenta adquirir el semáforo para lectura y si no lo obtiene el semáforo se suspende
int down_read_trylock(struct semaphore *sem)	Intenta adquirir un semáforo para lectura, pero con la peculiaridad de que nunca duerme, si el semáforo no está disponible devuelve un valor distinto de cero pero vuelve inmediatamente
int up_read(struct semaphore *sem)	Libera el semáforo para lectura
void down_write(struct rw_semaphore *sem)	Intenta adquirir el semáforo para escritura y si no lo obtiene el semáforo se suspende
int down_write_trylock(struct rw_semaphore *sem)	Intenta adquirir un semáforo para escritura, pero con la peculiaridad de que nunca duerme, si el semáforo no está disponible devuelve un valor distinto de cero pero vuelve inmediatamente
void up_write(struct rw_semaphore *sem)	Libera el semáforo para escritura
downgrade_writer()	Convierte atómicamente un semáforo adquirido para escritura en uno para lectura

Cuadro 5.10: Operaciones sobre variables semáforos de elctura escritura *rw_semaphore*

5.6. Deshabilitación de expropiación

Al tener un kernel expropiativo, un proceso del kernel puede parar su ejecución en cualquier momento para ceder el procesador a otro de mayor prioridad, esto significa que una tarea puede empezar a ejecutar la misma sección crítica que estaba ejecutando la que fue expropiada. Hay secciones de código tanto en el kernel de linux como en nuestro proyecto en las que no queda más remedio que evitar la expropiación, pero estas son las mínimas posibles. En nuestra implementación estas situaciones se dan cuando se realiza la modificación de parámetros de profiling a través de /proc y al hacer la cuenta de los experimentos.

preempt_disable()	Deshabilita la expropiación incrementando el contador de expropiaciones
preempt_enable()	Decrementa el contador de expropiaciones, comprobando si hay que replanificar y replanificando si este es cero
preempt_enable_no_resched()	Habilita la expropiación pero sin chequear
preempt_count()	Devuelve el contador de expropiaciones

Cuadro 5.11: Operaciones para gestionar la expropiación en el planificador

5.7. Deshabilitación local de interrupciones

Deshabiltar las interrupciones es uno de los mecanismos clave utilizado para asegurar que una secuencia de sentencias del kernel es tratada como una sección crítica. Permite que un proceso continue su ejecución aun cuando se active una señal de interrupción y de esta forma provee una manera efectiva de proteger estructuras de datos que también son accedidas por manejadores de interrupciones. Sin embargo, en sistemas multiprocesador, la deshabilitación de interrupciones locales no proteje contra accesos concurrentes de manejadores de interrupciones que se ejecuten en otras CPUs.

Para habilitar y deshabilitar se utlizan las funciones local_irq_disable( ) y local_irq_enable( ).

Para deshabilitar las interrupciones se pone a cero el flag IF del registro eflags, pero al final de la sección crítica no se puede simplemente activar el flag nuevamente. Esto es así porque los manejadores de interrupciones pueden ejecutarse de manera anidada, por lo tanto el kernel no conoce necesariamente cual era el valor del flag IF antes de que se ejecutara el hilo de ejecución actual. En estos casos, se debe guardar el valor anterior y luego restaurarlo a su valor original.

Para esto se utilizan las macros local_irq_save( ) y local_irq_restore( ). Típicamente estas macros se usan de la siguiente manera:

```
local\_irq\_save();
local\_irq\_disable( );
[...secci\'on cr\'itica]
local\_irq\_restore( );
```

5.8. Deshabilitación global de interrupciones

Algunas funciones críticas del kernel pueden ejecutarse en una CPU sólo si no se están ejecutando ningún manejador de interrupciones o función diferible en otras CPUs. Este requerimiento de sincronización se puede satisfacer deshabilitando las interrupciones globalmente. Sin embargo, este mecanismo disminuye el nivel de concurrencia global del sistema y se está dejando de usar ya que puede ser reemplazado por técnicas de sincronización más eficientes.

La deshabilitación global de interrupciones se realiza mediante la macro cli( ). Ensistemas monoprocesadores, esta macro simplemente se expande a _cli( ), deshabilitando las interrupciones locales. En sistemas multiprocesador, la macro espera hasta que terminen todos los manejadores de interrupciones y funciones diferibles, y luego toma un spinlock especial llamado global_irq_lock.Una vez que cli( ) vuelve, ningún manejador de interrupciones comenzará a ejecutarse hasta que las interrupciones sean reactivadas invocando la macro sti( ).

Capítulo 6

Comunicación mediante */proc*

El sistema de ficheros /proc es un mecanismos utilizado por el núcleo y por los módulos del núcleo para enviar información a los procesos. Es un sistema deficheros virtual que no está alojado en disco sino en moria principal.
Usaremos este pseudo sistema de ficheros para interactuar con las estructuras de los experimentos, acceder a sus estadísticas y consultar y configurar en caliente el comportamiento hyperthreading del procesador.

Para permitir la comunicación de nuestro sistema en ambas situaciones se ha creado en */ proc* un directorio adicional denominado *hyperthreading* al que añadiremos varias entradas adicionales. Estas entradas se añaden en forma de nodos virtuales o *v-nodos*.

Se crean así las siguientes entradas adicionales:

- **status**: leyendo el contenido de esta entrada:

  more /proc/hyperthreading/status

  se consulta el estado actual del planificador en cuanto al hyperthreading. Las posibles respuestas son:

  Escribiendo en esta entrada el usuario cambia el modo de funcionamiento del procesador. Así, según lo que escriba en ella haciendo:

  echo MODE > /proc/hyperthreading/experiments

valor obtenido	significado
auto_enabled	El planificador está funcionando en modo calidad de servicio (modo auto) y está habilitado el hyperthreading. Está en el estado HT_AUTO_ENABLED.
auto_disabled	El planificador está funcionando en modo calidad de servicio (modo auto) y está deshabilitado el hyperthreading. Está en el estado HT_AUTO_DISABLED.
force_enabled	El planificador no está funcionando en modo calidad de servicio está habilitado siempre el hyperthreading. Está en el estado HT_FORCE_ENABLED.
force_disabled	El planificador no está funcionando en modo calidad de servicio está deshabilitado siempre el hyperthreading. Está en el estado HT_FORCE_DISABLED.
waiting_timeslice	El planificador está funcionando en modo calidad de servicio y ha sido deshabilitado el hyperthreading para una tarea cuya calidad estaba siendo violada. Se ha detectado una considerable mejora en su rendimiento al eliminar el hyperthreading y se mantendrá sin él el planificador hasta que esta tarea finalice su timeslice.

Cuadro 6.1: Tabla de posibles estados del planificador leídos desde */proc*

se fuerza según el valor de MODE:

valor MODE	significado
auto	Se configura el planificador para que funcione en modo calidad de servicio.
enable	Se activa el hyperthreading.
disable	Se desactiva el hyperthreading.

Cuadro 6.2: Tabla de opciones de modo de uso configurables del planificador

- **experiments**: Esta entrada se ha usado como medio de depuración. En ella se puede leer una información distinta según la configuración previa que se haga sobre la misma. Al hacer:

  echo OPTION> /proc/hyperthreading/experiments

  se configura esta entrada para que muestre en función del valor de OPTION:

opción configurada	**resultado mostrado**
print_experiments	Muestra el número de experimentos de cada tipo que hay. Lo hace en este orden: número de bajo nivel local, número de bajo nivel global, número de alto nivel local y número de alto nivel global.
print_info _hl_experiments	Muestra los experimentos de alto nivel que hay insertados en el sistema.
print_statisticks	Muestra las estadísticas que hay almacenadas hasta ese momento. Para las locales muestras las de la tarea que está actualmente en el procesador.
reset_statisticks	Resetea todos losvalores estadísticos del sistema.
clear_experiments	Vacía el conjunto de experimentos que están insertados en el kernel. Se hace también un reseteo de las estadísticas del sistema que estaban referidas a dichos experimentos.
unset_profiling	Elimina el pid para el cual se esté haciendo el profiling de haber sido configurado previamente.

Cuadro 6.3: Tabla de opciones configurables para depuración de las estadísticas mediante */proc*

- **statistics**: Leyendo el contenido de esta entrada ("*more /proc/hyperthreading/statistics*") se vacia el triple buffer que estaba recogiendo los valores de profiling del proceso para el cuál se configuró.
  Recordemos que este buffer mostraba el pid del proceso y el valor de la medición puntual junto al id del experimento al que va referida. Además muestra también los resultados estadísticos globales.
  Escribiendo en esta entrada:

  echo PID > /proc/hyperthreading/statistics

  se configura el sistema para que haga profiling local para los experimentos insertados para el proceso de pid PID y para todos sus hijos.

- **sampling**: Leyendo el contenido de esta entrada:

  more /proc/hyperthreading/sampling

  se obtienen los valores de los parámetros relevantes en la calidad de servicio.

valores mostrados	**resultado mostrado**
samples_size	Indica el valor del tamaño de ventana que se está usando para calcular el valor medio de un experimento.
nticks	Indica cada cuántos ticks se actualizan los valores estadísticos.
min_ticks_out	Indica la cantidad mínima de ticks globales que hay que dejar pasar con el hyphyperthreading desactivado para que el proceso para el cual se produjo la violación remonte su comportamiento.
max_ticks_out	Indica el número máximo de ticks globales de reloj que hay que dejar pasar con el hyphyperthreading desactivado para que el proceso para el cual se produjo la violación remonte su comportamiento.

Cuadro 6.4: Tabla de descriptiva de la salida para la lectura de *sampling*

También se usará este nodo para configurar los valores anteriores. Así, con el comando

echo PARAMETER VALUE > /proc/hyperthreading/sampling

se configura con el valor VALUE el parámetro que diga PARAMETRO. Se recomienda ver la tabla 6.5.

parámetro modificado	**significado**
samples_size	Se modifica el tamaño de ventana para calcular la media.
nticks	Se cambia el número de ticks que se dejan pasar antes de realizar una nueva medición.
min_ticks_out	Se cambia el mínimo número de ticks que debe dejarse el hyperthreading desactivado antes de reconsiderar el comportamiento del procesos violado.
max_ticks_out	Se modifica el máximo número de ticks que ha de dejarse el hyperthreading desactivado para darle opción al proceso violado al recuperarse.

Cuadro 6.5: Tabla de parámetros configurables para qos mediante */proc*

- **__init __hta_proc_start()**

Esta llamada se ejecuta al arrancar el sistema. Como se puede apreciar en ella se crea el subdirectorio ***hyperthreading*** mediante la llamada *proc_mkdir*. Esta llamada nos devuelve la correspondiente entrada *parent* creada que es una estructura del tipo *proc_dir_entry*. En ella y con la llamada *create_proc_entry* se crean las entradas anteriormente mencionadas */proc/hyperthreading/status*, */proc/hyperthreading/statistics*, */proc/hyperthreading/experiments* y */proc/hyperthreading/sampling* .

Para cada una de ellas se completan adecuadamente los parámetros que apuntan a las funciones de lectura y de escritura.

```
int __init __hta_proc_start( void ){
    struct proc_dir_entry *parent=NULL;
    struct proc_dir_entry *proc_htasched=NULL;
    struct proc_dir_entry *proc_experiments=NULL;
    struct proc_dir_entry *statisticks_e=NULL;
    struct proc_dir_entry *sampling_e=NULL;
    printk( "HTA: Building proc entries... " );
if( cpu_has_ht ) {
parent = proc_mkdir( "hyperthreading", NULL );

proc_htasched = create_proc_entry( "status", 0666, parent );
proc_htasched->nlink = 1;
proc_htasched->data = (void*) NULL;
proc_htasched->read_proc = status_read_proc;
proc_htasched->write_proc = status_write_proc;

proc_experiments = create_proc_entry( "experiments", 0666, parent );
proc_experiments->nlink = 1;
proc_experiments->data = (void*) NULL;
proc_experiments->read_proc = experiments_read_proc;
proc_experiments->write_proc = experiments_write_proc;

statisticks_e = create_proc_entry( "statistics", 0666, parent );
statisticks_e->nlink = 1;
statisticks_e->data = (void*) NULL;
statisticks_e->read_proc = statistics_read_proc;
statisticks_e->write_proc = statistics_write_proc;

sampling_e = create_proc_entry( "sampling", 0666, parent );
sampling_e->nlink = 1;
```

```
sampling_e->data = (void*) NULL;
sampling_e->read_proc = sampling_read_proc;
sampling_e->write_proc = sampling_write_proc;

printk( "built.\n" );
} else
printk( "HT not available.\n" );
return 0;
}
```

Parte III

Resultados

Capítulo 7

Funcionalidades de la herramienta

7.1. Uso de la aplicación como profiler

El kernel propuesto permite realizar profiling a nivel de sistema que no requiere recompilaciones de kernel ni ningún módulo adicional.

Para realizar profiling con el hyperthreading desactivado hay que hacer previamente desde línea de comandos:

```
echo disable > /proc/hyperthreading/status
```

Si por el contrario se desea obtener las medidas con dos procesadores lógicos ha de hacerse:

```
echo enable > /proc/hyperthreading/status
```

Si se desea configurar el número de ticks que desea darse para espaciar la toma de medidas ha de hacerse un paso adicional (por defecto, el sistema lo hace cada tres ticks). Para un profiling con suficiente detalle podríamos poner cada 2 ticks de reloj.

```
echo ntiks 2 > /proc/hyperthreading/sampling
```

El profiling se puede realizar tanto a nivel global como para un proceso o conjunto específico de procesos determinado. Para cada una de estas opciones se permite la configuración dinámica de los experimentos a medir y otros parámetros. En concreto se permite:

1. **Configuración dinámica del tipo de profiling**.
   Este puede ser global o local a un proceso. En el caso del profiling local además se incluirán en los resultados finales las medidas de los hijos de tal proceso.

2. **Configuración dinámica de parámetros relevantes en el profiling**.
   Nos referimos al número de tiks *nticks* que se dejan pasar entre dos tomas de medidas consecutivas y al pid del proceso que se desea profilear en el caso de escoger profiling local.

3. **Configuración dinámica de los parámetros de rendimiento a estudiar**.
   Todos los posibles experimentos a estudiar y posibles modos se consideraron en el capítulo 5.8.

4. **Obtención de listado de resultados por procesos**.

5. **Obtención de listado de resultados locales**.

6. **Creación de gráficas de resultados**.

7.1.1. Pasos en la configuración para hacer un profiling global del sistema

PASO 1: Generación e inserción de los experimentos

El primer paso para realizar un profiling del sistema es generar un fichero *.xml* de configuración en el que se especifiquen las mediciones a realizar. Este xml puede ser generado automáticamente por nuestra herramienta Brank, aunque también puede escribirse a mano. A continuación puede verse un ejemplo de fichero de configuración:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE experiments SYSTEM "experiments.dtd">
<experiments>
  <experiment name="exp1">
    <ll_events>
      <local/>
      <global>
        <ll_event name="cache_misses_L1_p0" idx="3" type="pebs">
          <cnt>
            <escr address="0x3cc" value="0x12000204"/>
            <cccr address="0x36C" value="0x0003b000"/>
            <pmc msr_address="0x30C" pmc_address="12"/>
          </cnt>
          <pebs_config>
                <pebs_matrix_vert address="0x3f1" value="0x00000001"/>
                <pebs_enable_ia32 address="0x3f2" value="0x01000001"/>
          </pebs_config>
        </ll_event>
        <ll_event name="cycles_count_p0" idx="0" type="simple">
          <cnt>
            <escr address="0x3a2" value="0x26000204"/>
            <cccr address="0x360" value="0x0003d000"/>
            <pmc msr_address="0x300" pmc_address="0"/>
          </cnt>
        </ll_event>
        <ll_event name="instr_retired_p0" idx="2" type="simple">
          <cnt>
```

```
          <escr address="0x3b8" value="0x04001e04"/>
          <cccr address="0x36D" value="0x00039000"/>
          <pmc msr_address="0x30D" pmc_address="13"/>
        </cnt>
      </ll_event>
      <ll_event name="loads_retired_p0" idx="1" type="tagging">
        <cnt>
          <escr address="0x3cd" value="0x10000204"/>
          <cccr address="0x36E" value="0x0003b000"/>
          <pmc msr_address="0x30E" pmc_address="14"/>
        </cnt>
        <tag>
          <escr address="0x3bc" value="0x04000404"/>
          <cccr address="0x370" value="0x00035000"/>
          <pmc msr_address="0x310" pmc_address="16"/>
        </tag>
      </ll_event>
    </global>
  </ll_events>
  <hl_events>
    <local/>
    <global>
      <hl_event name="cache_miss_rate_L1_p0" idx="0" relation="op_rate">
        <event type="ll_global" idx="3"/>
        <event type="ll_global" idx="1"/>
        <parameters scale_factor="100" allowed_desviation="500"
                    threshold_type="dinamic_type_threshold"/>
        <actions if_return_up="act_none" if_return_down="act_none"
                    if_out_up="act_down_ht" if_out_down="act_down_ht"/>
      </hl_event>

      <hl_event name="ipc_p0" idx="1" relation="op_rate">
        <event type="ll_global" idx="2"/>
        <event type="ll_global" idx="0"/>
        <parameters scale_factor="100" allowed_desviation="500"
                    threshold_type="dinamic_type_threshold"/>
        <actions if_return_up="act_none" if_return_down="act_none"
                    if_out_up="act_down_ht" if_out_down="act_down_ht"/>
        </hl_event>

      <hl_event name="cicles_p0" idx="2" relation="op_none">
        <event type="ll_global" idx="0"/>
        <parameters scale_factor="1" allowed_desviation="500"
                    threshold_type="dinamic_type_threshold"/>
        <actions if_return_up="act_none" if_return_down="act_none"
                    if_out_up="act_down_ht" if_out_down="act_down_ht"/>
      </hl_event>

      <hl_event name="insts_p0" idx="3" relation="op_none">
        <event type="ll_global" idx="2"/>
        <parameters scale_factor="1" allowed_desviation="500"
                    threshold_type="dinamic_type_threshold"/>
        <actions if_return_up="act_none" if_return_down="act_none"
                    if_out_up="act_down_ht" if_out_down="act_down_ht"/>
      </hl_event>
    </global>
  </hl_events>
 </experiment>
</experiments>
```

Descripción del fichero de configuración

Puede observarse como en la primera parte (`<ll_events>`) se definen los experimentos de bajo nivel, que son los que también se podrían haber insertado con la herramienta *Brink & Abyss* [12] pese a que el proceso de generación del xml de configuración hubiera sido bastante más tedioso.
Si se observan los campos se notará que coinciden con los campos necesarios para la configuración de un experimento de bajo nivel como se ha comentado anteriormente.
La segunda parte (`<hl_events>`) es la que contiene los experimentos de alto nivel que es con la que el usuario decide realmente que es lo que quiere medir, puede ser simplemente el resultado de un evento de bajo nivel de los antes definidos (lo que se matiza con el tipo de relación op_none) o una relación entre eventos (tanto de alto nivel como de bajo nivel) permitiéndonos definir experimentos de la complejidad que queramos.

Inserción de los experimentos

```
?> ./parser_experiments experimentos_globales.xml
```

El xml obtenido en el apartado anterior se debe pasar como primer argumento al programa *parser_experiments* que realizará un parseado del *.xml* transformándolo a las estructuras de datos que maneja el kernel e insertándo los experimentos oportunos. El kernel al recibir estas estructuras realiza los chequeos que se enunciaron en el capítulo 3.5.3 para asegurarse de que todos los valores están en sus rangos válidos y que no hay ninguna incongruencia. Este chequeo no sería necesario en el caso de que la configuración de los experimentos se haya hecho desde la herramienta Brank. No obstante es necesario proporcionarlo puesto que la inserción se hace mediante una llamada al sistema y por tanto, cualquier usuario podría hacer una configuración perniciosa de los mismos provocando mediciones erroneas o incluso llevar al sistema a un kernel panic (si se ponen direcciones de registros inexistentes). No se pretende limitar la inserción de los experimentos a la aplicación Brank que hemos presentado puesto que son dos proyectos independientes, pero la recomendamos como medio absolutamente fiable en este sentido.

PASO 2: Medida de los resultados

Para consultar las estadísticas bastará consultar la entrada /proc/hyperthreading/statistics ejecutando por ejemplo el comando:

```
cat /proc/hyperthreading/statistics
```

En cada linea de la salida se especifica el identificador de la medición y el resultado de la medición.

Internamente las mediciones son almacenadas en un buffer, si este se llena dejan de poder añadirse más mediciones, por tanto para hacer un profiling correcto debemos de vaciar este buffer periódicamente a intervalos de tiempo que no haga que perdamos ninguna medida.

7.1.2. Pasos en la configuración para hacer profiling local a un proceso

PASO 1: Generación e inserción de los experimentos

El procedimiento será el mismo que en el caso de profiling global, pero teniendo en cuenta que en el *.xml* resultante tendremos los experimentos en la zona `<global>` en la de de `<local>` (`<ll_events>` e idénticamente con los de `<hl_events>`). En el ejemplo escrito anteriormente no aparecía ningún experimento de este tipo ya que todos eran globales. Cabe matizar que experimentos locales y globales pueden coexistir y se puede hacer profiling de ambos simultáneamente.
Como en el caso anterior hay que insertar dichos experimentos en el sistema haciendo:

```
?> ./parser_experiments experimentos_locales.xml
```

PASO 2: Activación del profiling para un determinado proceso

Para poder hacer profiling de un proceso determinado y no de manera global para todo el sistema como se había especificado antes, es necesario especificar el pid del proceso. Esto se hace volcando ese pid en la entrada /proc/hyperthreading/statistics, lo cual podría hacerse de la siguiente forma:

```
echo 12345 > /proc/hyperthreading/statistics
```

Siendo 12345 el pid del proceso a monitorizar. Destacamos que las medidas se realizarán tanto para el proceso indicado como para sus hijos.

PASO 3: Medida de los resultados

Para consultar las estadísticas bastará consultar la entrada /proc/hyperthreading/statisticks ejecutando por ejemplo el comando:

```
cat /proc/hyperthreading/statistics
```

Al hacer profiling local cada linea de la salida mostrará el pid del proceso monitorizado, el identificador de la medición y el resultado de la medición.

7.2. Configuración del sistema para mejorar la calidad de servicio

Para que el sistema este actuando en modo Calidad de Servicio no es necesario deshabilitar la funcionalidad de profiling. De hecho, aunque el usuario esté ingnorándolo, el algoritmo por debajo está tomando mediciones idénticas a las pedidas en profiling para poder razonar sobre la necesidad de la activación / desactivación del hyperthreading.

PASO 1: Configurarción los parámetros influyentes en el algoritmo

En primer lugar es necesario configurar los parámetros que son básicos en las decisiones del algoritmo. Proponemos, por ejemplo, la configuración siguiente:

```
1:   echo auto > /proc/hyperthreading/status
2:   echo ntiks 3 > /proc/hyperthreading/sampling
3:   echo min_ticks_out 3 > /proc/hyperthreading/sampling
4:   echo max_ticks_out 20 > /proc/hyperthreading/sampling
5:   echo samples_size 5 > /proc/hyperthreading/sampling
```

La línea 1 pone el planificador en modo *auto* para que el algortimo de Calidad de Servicio esté activo. Después desde la línea 2 a la 5 se configuran los parámetros del algoritmo que se explicaron en el capítulo 3.5.3.

PASO 2: Generación e inserción del experimento que orientará la calidad de servicio

Debemos especificar el experimento en base al cual se realizará la calidad de servicio (tasa de fallos de primer nivel, tasa de fallos de segundo nivel, ipc etc..). Este *.xml* de nuevo puede ser generado automáticamente por nuestra herramienta Brank, aunque también puede escribirse a mano. A continuación puede verse un ejemplo de fichero de configuración en el que se ha elegido el *ipc* como experimento para estudiar el rendimiento:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE experiments SYSTEM "experiments.dtd">
<experiments>
  <experiment name="exp1">
```

```
    <ll_events>
      <local>
        <ll_event name="cycles_count_p0" idx="0" type="simple">
          <cnt>
            <escr address="0x3a2" value="0x26000204"/>
            <cccr address="0x360" value="0x0003d000"/>
            <pmc msr_address="0x300" pmc_address="0"/>
          </cnt>
        </ll_event>
        <ll_event name="instr_retired_p0" idx="1" type="simple">
          <cnt>
            <escr address="0x3b8" value="0x04001e04"/>
            <cccr address="0x36D" value="0x00039000"/>
            <pmc msr_address="0x30D" pmc_address="13"/>
          </cnt>
        </ll_event>
      </local>
      <global/>
    </ll_events>
    <hl_events>
      <local>
        <hl_event name="ipc_p0" idx="0" relation="op_rate">
          <event type="ll_local" idx="1"/>
          <event type="ll_local" idx="0"/>
          <parameters scale_factor="100" allowed_desviation="500"
                      threshold_type="dinamic_type_threshold"/>
          <actions if_return_up="act_none" if_return_down="act_none"
                      if_out_up="act_down_ht" if_out_down="act_down_ht"/>
       </hl_event>
      </local>
      <global/>
    </hl_events>
  </experiment>
</experiments>
```

Se han introducido el número de instrucciones retiradas (*instr_retired_p0*) y el número de ciclos (*cycles_count_p0*) como experimentos para medir para cada tarea activa en el sistema. Se medirán solo para el procesador lógico cero. El factor de escala por el que queremos que multiplique la división de intrucciones entre ciclos es 100 (para obtener la máxima precisión posible). Además, se permitirá una desviación de 500 en torno al valor de la media que salga (este tratamiento de los umbrales se especificó en el capítulo 3.5.3).

Para observar una mejora del rendimiento basta con consultar el tiempo de ejecución del proceso lanzado comparándolo con el de su lanzamiento en otro kernel.

Capítulo 8

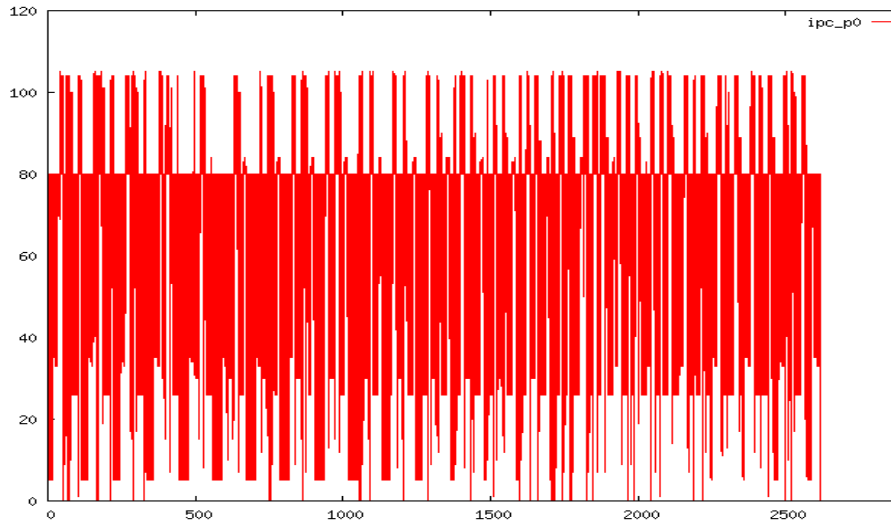
Resultados

8.1. Benchmarking

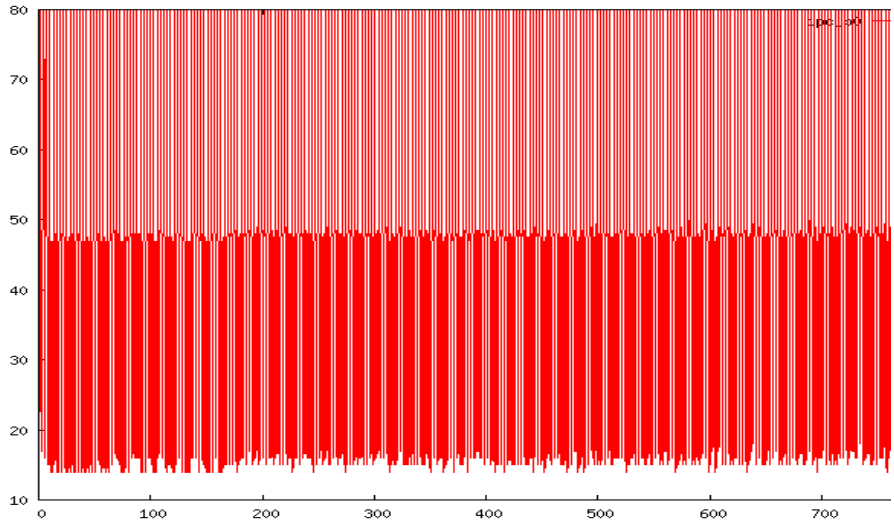
Para poder decidir cuando un proceso está siendo afectado negatívamente por otro debido a la comportición de los recursos la estrategia comentada anteriormente se basa en estudiar la curva de rendimiento a lo largo del tiempo de ejecución del proceso y decidir en cada momento si favorece al más prioritario desactivar HT (en cuyo caso lo desactivaremos) o si la pérdida de rendimiento es causa del comportamiento natural del proceso (en cuyo caso desactivar no nos favorece en nada, lo único que hace es empeorar el rendimiento global).

Para evaluar el comportamiento natural de los procesos utilizamos nuestro prototipo con su funcionalidad como profiler, se ha medido la evolución del ipc (multiplicado por 100 para tener un valor más preciso) a lo largo de la ejecución de los procesos que luego se van a utilizar para establecer calidad de servicio. Puesto que el comportamiento ideal del proceso sería que este estuviera sólo en la CPU, las pruebas se han realizado con el hyperthreading desactivado y lanzando los procesos de manera secuencial (los procesos elegidos para el benchmarking provieen de los SPECINT2000 y SPECFP2000).
Estos benchmarks son:

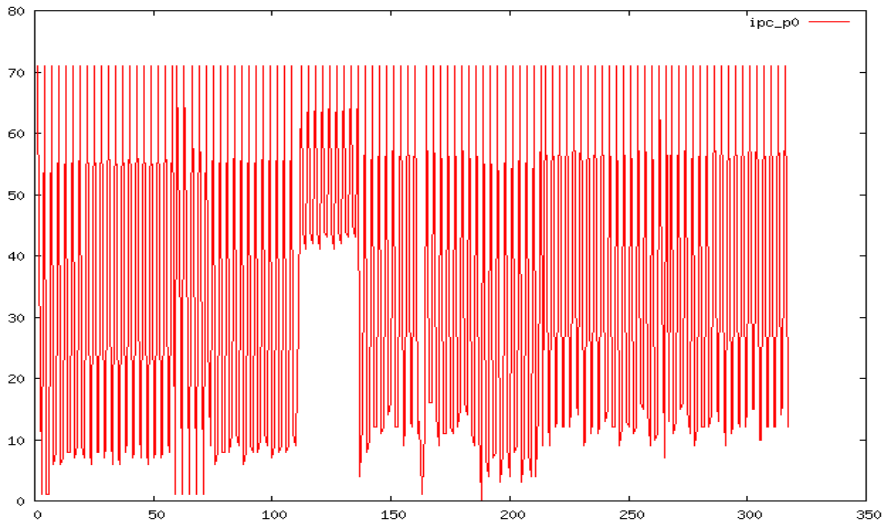
8.1.1. apsi



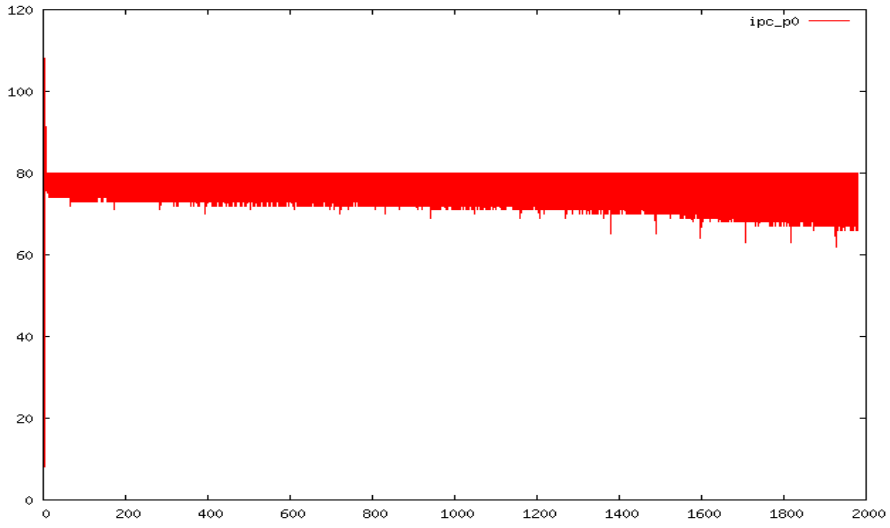
8.1.2. art



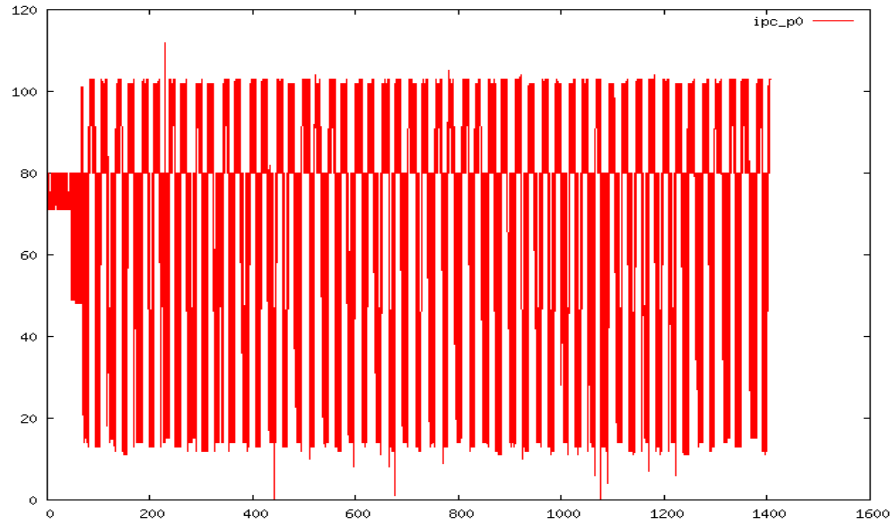
8.1.3. gzip



8.1.4. twolf



8.1.5. wupwise



De estas gráficas y a través de estudios experimentales, la desviación de IPC que usaremos en las pruebas es:

1. Apsi: 30
2. Art: 20
3. Gzip: 30
4. Twolf: 8
5. Wupwise: 20

Además en las pruebas de calidad de servicio se han configurado los siguentes parámetros:

1. samples_size 8 (tamaño del buffer circular que almacena los últimos valores de ipc)
2. nticks 5 (número de ticks cada el que se actualizan las mediciones)
3. min_ticks_out 5
4. max_ticks_out 30 (estos 2 últimos parámetros controlan el rango de ticks en que se evalúa el ipc del proceso para hacer calidad de servicio). La configuración y consulta de estos valores se puede consultar y modificar a través de /proc cómo se explica en el capítulo correspondiente.

8.2. Calidad de servicio

A partir de las gráficas anteriores podemos estimar la desviación que experimenta el ipc respecto al valor medio cuando el proceso no está siendo "molestado"por ningún otro, así si al tener el hyperthreading activado el ipc se desvía más de la cuenta podemos concluir que es por culpa del otro proceso.

Si las pruebas se hacen sin priorizar a ningún proceso no se afecta sustancialmente al rendimiento, sólo se consiguen oscilaciones de menos 0.5 % que podemos considerar despreciables.

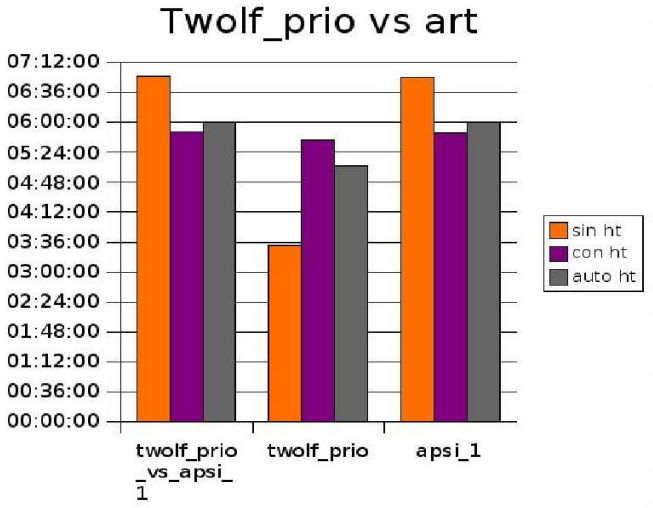
La idea es conseguir un aumento del rendimiento local del proceso más prioritario sin influir negátivamente en el rendimiento global y en el de los otros procesos , lo que se conoce con el nombre de políticas de tiempo real suave.

Las gráficas mostradas siguen todas la misma estructura, 3 grupos de barras cada uno de ellos con 3 barras, el grupo de la izquierda muestra los resultados de tiempo global (desde que empiezan los 2 procesos hasta que acaban los 2 procesos), y los otros 2 grupos las estadísticas para cada uno de los procesos (el nombre del proceso correspondiente se indica en la base de la columna). De las 3 barras de cada grupo la barra de más a la izquierda indica el tiempo medido con el hyperthreading desactivado, la barra de en medio el tiempo medido con el hyperthreading activado y la de la derecha el tiempo medido con nuestra implementación. Cabe destacar que las medidas con y sin hyperthreading han sido tomadas con una versión del kernel "limpia", sin ningún añadido por nuestra parte ni parche de SMT, símplemente configuranco la activación/-desactivación del hyperthreading desde la BIOS.

Podríamos dividir en 2 los tipos de resultados obtenidos que hemos obtenido: - Aquellos en los que la diferencia en tiempos a nivel global (grupo de columnas a la izquierda) sin hyperthreading y con hyperthreading (columnas izquierda y central) es pequeña, lo que significa que los procesos no se interfieren entre sí y que nos convendría tener el hyperthreading activado. En este caso la repercusión de nuestro prototipo será mínima. - Aquellos en los que la diferencia es grande. En este caso sí podemos evitar que los procesos se interfieran entre sí.

A continuación se muestran los resultados de los experimentos:

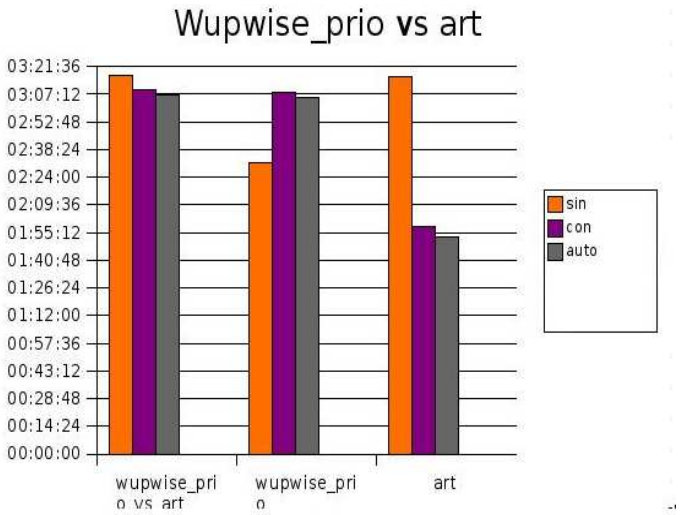
8.2.1. Twolf_prio vs art



	sin ht	con ht	auto ht	Incremento Rendimiento
twolf_prio_vs_apsi_1	06:54:57	05:47:39	06:00:42	-3.87
twolf_prio	03:31:46	05:38:18	05:07:04	9.32
apsi_1	06:53:54	05:46:38	05:59:41	-3.88

Este ejemplo pertenece al segundo grupo antes comentado se aprecia como la diferencia de tiempos globales es considerable, por tanto sí que podemos favorecer al proceso que queramos, en este caso a Twolf. Se ha conseguido un incremento del rendimiento para twolf del 9.33 % respecto a la ejecución con hyperthreading activado, aunque se ha deteriorado el rendimiento global en un 3.87 % y el del proceso apsi en un 3.88 %.

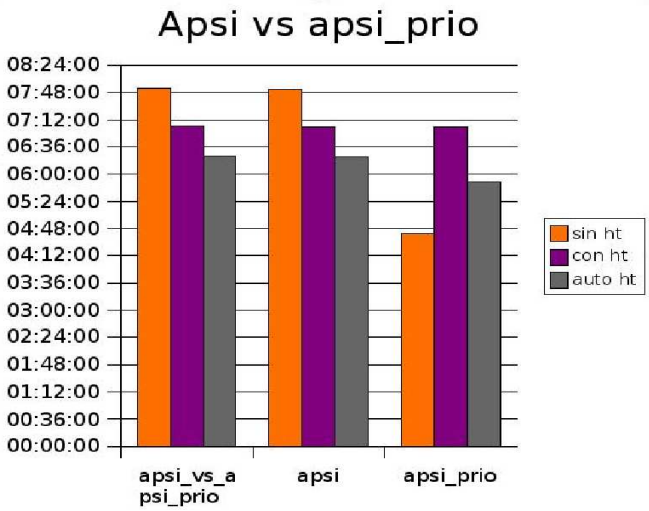
8.2.2. Wupwise_prio vs art



	sin ht	con ht	auto ht	Incremento Rendimiento
wupwise_prio_vs_art	03:17:20	03:09:12	03:06:32	1.69
wupwise_prio	02:31:18	03:08:11	03:05:30	1.70
art	03:16:18	01:58:29	01:52:44	4.62

Este experimento podríamos considerarlo dentro del primer grupo, ya que la diferencia de tiempos globales es muy pequeña. Aun así se consigue mejorar el rendimiento de wupwise (el más prioritario) en un 1.7 %, cabe destacar que en este caso también se incrementa el rendimiento el rendimiento del otro proceso (art) en un 4.62 % y el rendimiento global en un 1.69 %. Esto es debido a que en ocasiones hacer que el proceso más prioritario esté sólo en la CPU cuando pierde rendimiento redunda en un incremento del rendimiento del otro proceso, ya que el prioritario tampoco molesta al otro.

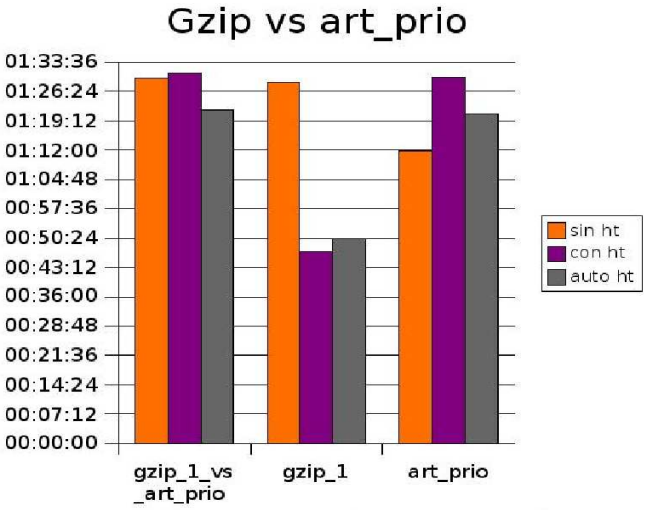
8.2.3. Apsi vs Apsi_prio



	sin ht	con ht	auto ht	Incremento Rendimiento
apsi_vs_apsi_prio	07:53:13	07:03:40	06:23:59	9.40
apsi	07:52:12	07:02:39	06:22:58	9.42
apsi_prio	04:42:02	07:02:38	05:49:54	17.24

Este es un caso extremo del segundo grupo, la diferencia de tiempos globales es bastante grande (en torno al 11 %). Nuestro prototipo obtiene un incremento del rendimiento en el proceso prioritario del 17.24 % y no sólo eso sino que además produce un incremento del 6.22 % en el otro proceso y del un 6:23 % en el global (es un caso similar al contado antes en el que favorecer a un proceso repercute positívamente en el rendimiento del otro).

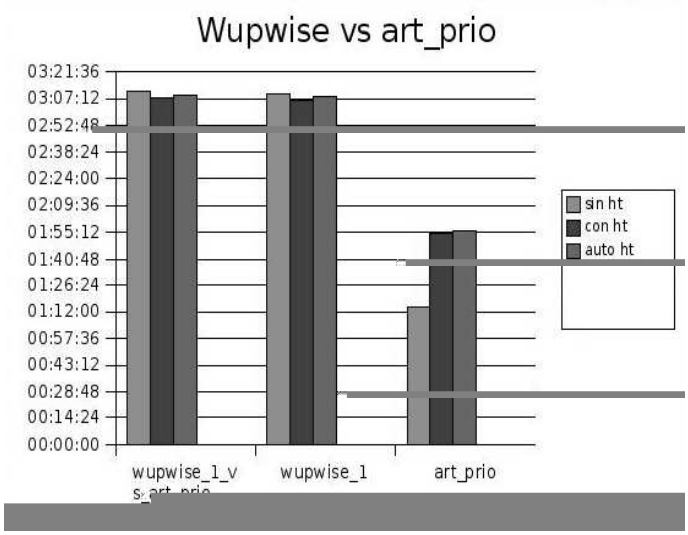
8.2.4. Gzip vs art_prio



	sin ht	con ht	auto ht	Incremento Rendimiento
gzip_1_vs_art_prio	01:29:37	01:30:50	01:21:48	9.96
gzip_1	01:28:36	00:47:10	00:50:16	-7.40
art_prio	01:11:52	01:29:49	01:20:46	10.09

Se trata de otro ejemplo del segundo grupo, pero en este caso pese a que se consigue una mejora del rendimiento del 10.09 % en el proceso más prioritario y un 9.96 % más en el global en el otro proceso se pierde un 7.40 % de rendimiento.

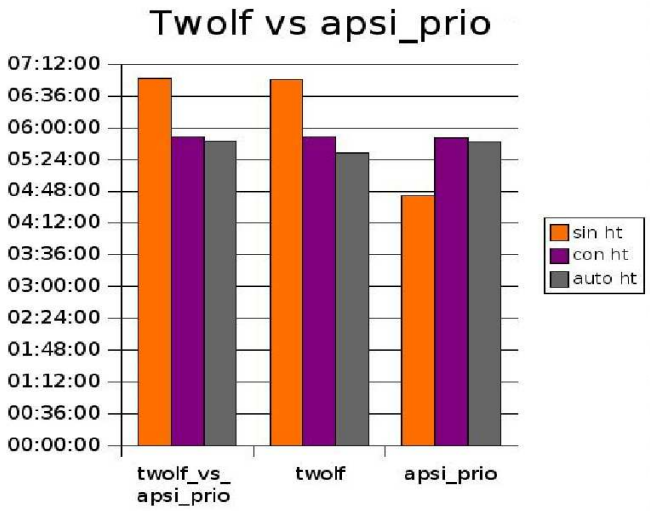
8.2.5. Wupwise vs art_prio



	sin ht	con ht	auto ht	Incremento Rendimiento
wupwise_1_vs_art_prio	03:11:04	03:07:39	03:08:17	-.63
wupwise_1	03:10:03	03:06:37	03:07:16	-.63
art_prio	01:14:52	01:54:29	01:54:20	.07

Se trata de otro ejemplo del primer grupo, pero en este caso el intento de mejora incurre en una mínima pérdida de rendimiento en ambos procesos y en el global.

8.2.6. Twolf vs apsi_prio



	sin ht	con ht	auto ht	Incremento Rendimiento
twolf_vs_apsi_prio	06:56:23	05:50:00	05:45:22	1.25
twolf	06:55:22	05:48:59	05:31:26	4.97
apsi_prio	04:43:23	05:48:24	05:44:21	1.15

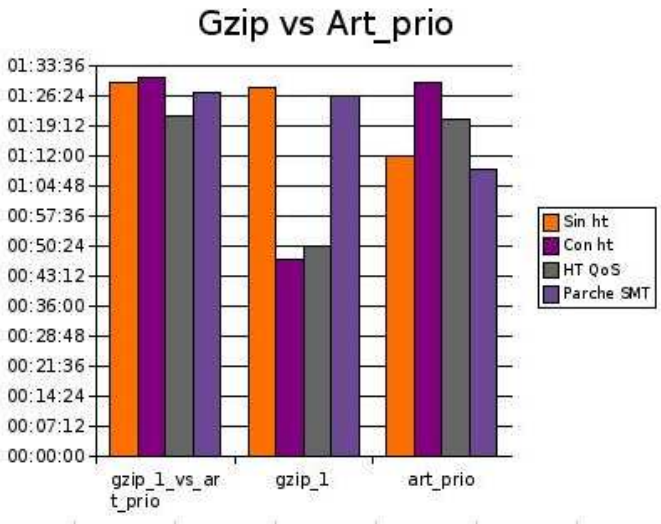
Otro ejemplo más del segundo grupo, en esta prueba se consigue un incremento del 1.15 % en el más prioritario, lo cual repercute en un incremento de 4.97 % en el otro proceso y de 1.25 % en el global

8.3. Comparativa con el parche de SMT

También se ha realizado pruebas comparando el resultado de nuestro parche y el otro existente para SMT, se puede comprobar como el otro proporcina un incremento del rendimiento algo mayor en el proceso más prioritario, pero repercute en una bajada brutal del rendimiento para el otro proceso y a nivel global. Veamos un par de gráficas que los muestran, sin embargo nuestro parche aunque consigue un incremento del rendimiento menor en el proceso prioritario, mantiene el rendimiento o incluso lo incrementa para el global y para el otro proceso.

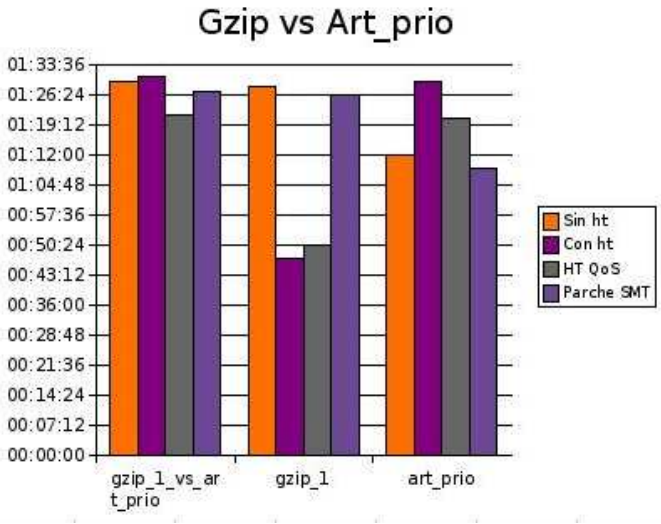
Las gráficas siguen con la misma estructura antes comentada, salvo que se ha añadido una nueva columna (la cuarta de cada grupo) en la que se muestra el tiempo empleado con el parche de SMT.

8.3.1. Gzip vs Art_prio SMT



	Sin ht	Con ht	HT QoS	Parche SMT	Ganancia QoS Con ht	Ganancia Parche Smt Con Ht
gzip_1 vs art_prio	01:29:37	01:30:50	01:21:48	01:27:18	9,96	3,66
gzip_1	01:28:36	00:47:10	00:50:16	01:26:17	-7,4	-84,51
art_prio	01:11:52	01:29:49	01:20:46	01:08:59	10,09	23,35

8.3.2. Twolf_prio vs Equake SMT



	Sin ht	Con ht	HT QoS	Parche SMT	Ganancia QoS-Con ht	Ganancia Parche Smt Con Ht
twolf_prio vs equake	04:25:49	04:13:35	04:18:28	04:23:02	-1,78	-3,65
twolf_prio	03:41:13	04:12:34	04:17:27	03:36:50	-1,79	14,2
equake	04:24:47	02:08:14	02:08:33	04:21:59	0,16	-104,1

A continuación se muestra una tabla resumen con más medidas tomadas:

	Sin ht	Con ht	HT QoS	Parche SMT	Ganancia QoS-Con ht	Ganancia Parche Smt Con Ht
gzip_1 vs art_prio	01:29:37	01:30:50	01:21:48	01:27:18	9,96	3,66
gzip_1	01:28:36	00:47:10	00:50:16	01:26:17	-7,4	-84,51
art_prio	01:11:52	01:29:49	01:20:46	01:08:59	10,09	23,35
gzip_1 vs art	01:32:18	01:22:20	01:26:04	01:22:39	-4,67	-0,23
gzip_1	00:57:11	00:46:41	00:47:12	00:47:08	-0,66	-1,44
art	01:31:16	01:21:18	01:25:02	01:21:37	-4,73	-0,23
gzip_1 vs gzip_2	00:58:14	00:48:49	00:48:32	00:48:17	1,17	0,65
gzip_1	00:57:05	00:47:16	00:47:20	00:47:15	0,76	0,02
gzip_2	00:57:12	00:47:48	00:47:30	00:47:07	1,22	0,86
gzip_1 vs gzip_prio	00:58:18	00:48:35	00:49:03	00:56:22	-0,57	-15,44
gzip_1	00:57:17	00:47:33	00:48:02	00:55:20	-0,61	-15,78
gzip_prio	00:34:33	00:47:11	00:45:39	00:31:42	3,65	33,3
gzip_prio vs art	01:30:57	01:25:22	01:26:00	01:33:04	-1,39	-9,21
gzip_prio	00:34:27	00:47:05	00:47:16	00:32:24	-0,23	31,74
art	01:29:54	01:24:20	01:25:38	01:32:02	-1,4	-9,33
twolf_prio vs apsi_1	06:54:57	05:47:39	06:00:42	06:45:32	-3,87	-16,81
twolf_prio	03:31:46	05:38:18	05:07:04	03:24:13	9,32	39,63
apsi_1	06:53:54	05:46:38	05:59:41	06:44:30	-3,88	-16,85
twolf_prio vs equake	04:25:49	04:13:35	04:18:28	04:23:02	-1,78	-3,65
twolf_prio	03:41:13	04:12:34	04:17:27	03:36:50	-1,79	14,2
equake	04:24:47	02:08:14	02:08:33	04:21:59	0,16	-104,1
twolf vs apsi_1	07:00:13	05:45:32	05:47:19	05:42:52	-0,65	0,69
twolf	06:04:06	05:31:10	05:42:11	05:30:14	-3,2	0,41
apsi_1	06:59:11	05:44:31	05:46:18	05:41:50	-0,66	0,7
twolf vs apsi_prio	06:56:23	05:50:00	05:45:22	03:01:21	1,25	48,28
twolf	06:55:22	05:48:59	05:31:26	03:00:19	4,97	48,42
apsi_prio	04:43:23	05:48:24	05:44:21	00:00:01	1,15	99,99
twolf vs equake_prio	04:18:47	04:14:15	04:21:45	04:17:36	-2,87	-1,26
twolf	04:17:45	04:13:14	04:20:44	04:16:34	-2,88	-1,26
equake_prio	02:41:15	02:08:38	02:08:29	01:29:08	0,07	30,7

Cuadro 8.1: Tabla de resultados de los experimentos

	Sin ht	Con ht	HT QoS	Parche SMT	Ganancia QoS-Con ht	Ganancia Parche Smt Con Ht
twolf vs equake	04:23:52	04:20:59	04:15:31	04:12:35	2,17	3,31
twolf	04:22:51	04:19:58	04:14:29	04:11:34	2,19	3,32
equake	02:39:00	02:08:26	02:08:39	02:08:06	-0,1	0,15

Cuadro 8.2: Tabla de resultados de los experimentos

Capítulo 9

Conclusiones

Según la ley de Moore la velocidad de los procesadores se dobla cada 18 meses. No obstante, parece ser (atendendiendo a las predicciones realizadas por especialistas en el tema), que esta ley dejará de cumplirse en unos 12 años, debido a las limitaciones físicas de la tecnología actual.

Los dos mecanismos propuestos para solucionar el problema son el paralelismo y la especialización, unidos a un software que permita utilizar los procesadores de manera eficiente. La especialización consiste en ampliar la arquitectura con extensiones especializadas en distintas tareas de procesamiento. El paralelismo es la opción más explotada de conseguir mejoras de productividad minimizando la inversión en hardware especializado.

Actualmente existen diversas alternativas en el contexto de la explotación del paralelismo en un computador. Principalmente han cobrado importancia dos: los diseños basados en varios procesadores en un chip (MoC)y el diseño fundamentado en un único procesador que permite la ejecución simultánea de varios hilos (*Multithreading Simultáneo* ó SMT).

El SMT es la opción más sofisticada de las 2. Actualmente no hay ninguna extensión del planificador de procesos de Linux que explote al máximo las capacidades aportadas por SMT. Siguiendo esta linea de investigación, hemos desarrollado un planificador para procesadores de la familia Pentium IV con tecnologá HyperThreading que tiene en cuenta las características arquitectónicas subyacentes.

Los resultados obtenidos –7.2– muestran que es posible **mejorar el rendimiento** de tareas de tiempo real suave (ver 2.4.3) **sin influir negativamente en el rendimiento global** del sistema, mejorando la implementación actual del planificador SMP (que no

tiene en cuenta las características concretas de la arquitectura).
Se ha conseguido que el trato preferente por prioridad a los procesos (calidad de servicio) esté soportado por el planificador.

Queremos destacar, por último, que alcanzar estos resultados no ha sido una tarea fácil.
Linux es un sistema operativo real. Por ello abordar una tarea de desarrollo sobre él es una labor muy compleja debida a varios factores. Uno de ellos es la gran extensión de su código fuente, en cuyo interior hay fragmentos que han sido sometidos a un intenso proceso de optimización a lo largo de los años. Esta optimización va en decremento de la legibilidad y claridad de la implementación, sobre todo si tenemos en cuenta la casi ausencia de comentarios a lo largo del código.
Por otra parte se requiere la comprensión completa del código del núcleo para poder desarrollar extensiones o modificaciones en él. Esta visión global no permanece estática durante mucho tiempo, ya que la comunidad de desarrollo del kernel libera versiones nuevas constantemente.

La tarea de depuración de las aplicaciones en modo usuario es una árdua tarea, sin embargo existen muchas herramientas para realizar esta tarea de manera sencilla. No ocurre lo mismo cuando lo que se intenta depurar es el código de un sistema operativo como Linux, ya que no existen mecanismos de depuración comparables. Un error del sistema puede llevar a tener que resetear la máquina. Además requiere recompilar el kernel entero tarea que puede llevar hasta unos 25 minutos.

Apéndice A

Instalación del prototipo

A.1. Descargar archivos fuentes

En primer lugar debemos descargar los archivos fuentes del kernel, y en concreto de la versión sobre la que queramos añadir el parche. El trabajo presentado se ha realizado sobre la versión *2.6.13* pero puede aplicarse a versiones futuras siempre que no sufra modficaciones relevantes en lo que al planificador se refiere.
Para realizar la descarga se recomienda acudir a *http://www.kernel.org*.
Podemos escoger entre descargar el *.tar.gz* o descargar el *.tar.bz*2 como haremos en el ejemplo mostrado.
Copiaremos el archivo que hemos descargado a: */usr/src*

```
cp linux-2.6.13.tar.bz2 /usr/src/
```

Lo descomprimimos (en el caso del *.bz*2 hacer)

```
cd /usr/src/
bunzip2 linux-2.6.13.tar.bz2
tar -xf linux-2.6.13.tar
```

A.2. Aplicamos el parche

Entraremos en el directorio que se ha creado al descomprimir.

```
cd linux-2.6.13
```

Copiamos el parche al directorio en el que se descomprimieron las fuentes y hacemos:

```
cd linux-2.6.13+\newline
~/linux-2.6.13$ patch -p1 < ../patch-2.6.13-htqos.patch
```

A.3. Configurando el kernel

La configuración del kernel en Linux se basa en un archivo */usr/src/linux/.config*, de crítico contenido. Existen no obstante interfaces más comprensibles basadas en el comando make:

make config
Pide una respuesta para cada opción de configuración. No permite corrección y en general es menos flexible que el resto.

make menuconfig
Presenta las opciones de configuración en forma de menú con interfaz gráfica en terminal de caracteres, clasificadas por grupos y con ayuda explicativa sobre cada una. Cada opción puede compilarse en el kernel (marcada con *) o cargarse como módulo (marcada con M). Cuando la opción sólo puede tomarse compilada la propia interfaz así lo hace.

make xconfig
Como make menuconfig pero en X-Windows. Requiere autorización para root en el uso del ambiente gráfico.

Escogeremos la segunda opción y hacemos

```
~/linux-2.6.13$ make menuconfig
```

En principio dejaremos las opciones que aparezcan activadas por defecto.
Entramos en las opciones *Processor type and features* para activar nuestro parche.
Seleccionamos la familia de procesadores *Pentium 4* y activamos la opción del parche..
Al terminar salimos salvando los cambios realizados.

A.4. Compilación del kernel

Si no se tiene instalado un sistema operativo *Debian GNU/Linux* se debe seguir el siguiente paso desde el directorio de los fuentes:

```
~/linux-2.6.13$ make
```

y esperar a que se termine la compilación para poder instalar.
Si por lo contrario se desea crear un paquete Debian ha de hacerse

```
~/linux-2.6.13$ make-kpkg kernel_image --version 1
```

A.5. Instalación de Linux

Si se escogió la primera opción en el paso anterior deben instalarse los módulos, copiar al directorio de arranque el mapa de memoria del kernel, copiar la imagen del kernel compilada a */boot* y añadir una entrada en el gestor de arranque (GRUB o LILO).

```
make modules_install
cp linux-2.6.13/arch/i386/boot/bzImage /boot
cp linux-2.6.13/System.map /boot
```

Para crear una entrada adicional en el gestor de arranque hay que hacer e el caso de Grub:

```
update-grub
```

En el caso de Lilo hay que editar el archivo */etc/lilo.conf*

```
image=/boot/bzImage
label=linux 2.6.13-htqos
read-only
append="root=LABEL=/"
```

Tras añadir la entrada, es suficiente con ejecutar lilo en la linea de comandos para que se grabe el nuevo menú en el MBR.

Si se escogió la segunda opción en el paso anterior, basta hacer

```
\begin{verbatim}
~$ dpkg -i kernel-image-2.6.13-htqos_1_i386.deb
```

Bibliografía

[1] Paper sobre los contadores de rendimiento en los procesadores Pentium 4. *IEEE micro Pentium 4 Performance Monitoring Features* Brinkley Sprunt, Bucknell University

[2] Página oficial del programa *Performance Application Programming Interface* http://icl.cs.utk.edu/papi/

[3] *Linux Kernel Developement* Robert Love, Editorial Sams Publishing

[4] *RTLinux gpl FTP repositorio* www.rtlinux-gpl.org

[5] *RTLinux HOW TO sobre compilación e instalación. Página de referencia para consultas* www.faqs.org/docs/Linux-HOWTO/RTLinux-HOWTO.html

[6] *Linux Device Drivers 3rd Edition* Editorial OŔeillyś, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

[7] *Concurrencia y sincronización en el kernel de Linux* http://linux.linti.unlp.edu.ar/informes/concurrencia-sincronizacion-linux.pdf

[8] *IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide*, Order Number: 253668, http://developer.intel.com

[9] *IA-32 Intel Architecture Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z*, Order Number: 253667i http://developer.intel.com

[10] *IA-32 Intel Architecture Optimization. Reference Manual*, Order Number: 248966-011 http://developer.intel.com

[11] *Article:Pentium 4 Performance monitoring feaures* Brinkley Sprunt, Artículo. Bucknell University.

[12] Página oficial del programa *Brink and Abyss Pentium 4 Performance Counter Tools For Linux*
www.eg.bucknell.edu/ bsprunt/emon/brink_abyss/brink_abyss.shtm

[13] *Sistemas Operativos* William Stallings. Cuarta edición. Prentice Hall, 2001.

[14] *Symbiotic Jobscheduling for a Simultaneous Multithreading Processor*, Dean M. Tullsen & Allan Snavely, in Procceding of ASPLOS IX, November 2000.

[15] Página de documentación de Debian,
http://www.debian.org/doc/